ALGORITHMIQUE ET PROGRAMMATION EN JAVA

Algeria-Educ.com

Consultez nos catalogues sur le Web

www.dunod.com

ALGORITHMIQUE ET PROGRAMMATION EN JAVA

Cours et exercices corrigés

Vincent Granet
Maître de conférences à l'ESINSA
(École supérieure d'ingénieurs
de l'université de Nice-Sophia Antipolis)

2e édition

DUNOD

Illustration de couverture :
Digital Vision

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.
Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).

© Dunod, Paris, 2000, 2004
ISBN 2 10 0048463 X

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

à Maud

Table des matières

AVANT-PROPOS XV

CHAPITRE 1 • **INTRODUCTION** 1
1.1 Environnement matériel 1
1.2 Environnement logiciel 3
1.3 Les langages de programmation 4
1.4 Construction des programmes 10
1.5 Démonstration de validité 11

CHAPITRE 2 • **ACTIONS ÉLÉMENTAIRES** 13
2.1 Lecture d'une donnée 13
2.2 Exécution d'une procédure prédéfinie 14
2.3 Écriture d'un résultat 15
2.4 Affectation d'un nom à un objet 15
2.5 Déclaration d'un nom 16
2.5.1 Déclaration de constantes 16
2.5.2 Déclaration de variables 17
2.6 Règles de déduction 17
2.6.1 L'affectation 17
2.6.2 L'appel de procédure 18
2.7 Le programme sinus écrit en Java 18
2.8 Exercices 20

CHAPITRE 3 • **TYPES ÉLÉMENTAIRES** 21
3.1 Le type entier 22
3.2 Le type réel 23

3.3 Le type booléen 26
3.4 Le type caractère 27
3.5 Constructeurs de types simples 29
3.6 Exercices 30

CHAPITRE 4 • **EXPRESSIONS** 31
4.1 Évaluation 32
4.1.1 Composition du même opérateur plusieurs fois 32
4.1.2 Composition de plusieurs opérateurs différents 32
4.1.3 Parenthésage des parties d'une expression 33
4.2 Type d'une expression 33
4.3 Conversions de type 34
4.4 Un exemple 34
4.5 Exercices 37

CHAPITRE 5 • **ÉNONCÉS STRUCTURÉS** 39
5.1 Énoncé composé 39
5.2 Énoncés conditionnels 40
5.2.1 Énoncé `choix` 40
5.2.2 Énoncé `si` 42
5.3 Résolution d'une équation du second degré 43
5.4 Exercices 46

CHAPITRE 6 • **PROCÉDURES ET FONCTIONS** 47
6.1 Intérêt 47
6.2 Déclaration d'un sous-programme 48
6.3 Appel d'un sous-programme 49
6.4 Transmission des paramètres 50
6.4.1 Transmission par valeur 51
6.4.2 Transmission par résultat 51
6.5 Retour d'un sous-programme 51
6.6 Localisation 52
6.7 Règles de déduction 54
6.8 Exemples 55

CHAPITRE 7 • **PROGRAMMATION PAR OBJETS** 61
7.1 Objets et classes 61
7.1.1 Création des objets 62
7.1.2 Destruction des objets 62
7.1.3 Accès aux attributs 63
7.1.4 Attributs de classe partagés 64
7.1.5 Les classes en Java 64
7.2 Les méthodes 65

7.2.1 Accès aux méthodes 66
7.2.2 Constructeurs 66
7.2.3 Les méthodes en Java 67
7.3 Assertions sur les classes 68
7.4 Exemples 69
7.4.1 Équation du second degré 69
7.4.2 Date du lendemain 72
7.5 Exercices 75

CHAPITRE 8 • **ÉNONCÉS ITÉRATIFS** 77
8.1 Forme générale 77
8.2 L'énoncé `tantque` 78
8.3 L'énoncé `répéter` 79
8.4 Finitude 80
8.5 Exemples 80
8.5.1 Factorielle 80
8.5.2 Minimum et maximum 81
8.5.3 Division entière 81
8.5.4 Plus grand commun diviseur 82
8.5.5 Multiplication 83
8.5.6 Puissance 83
8.6 Exercices 84

CHAPITRE 9 • **LES TABLEAUX** 87
9.1 Déclaration d'un tableau 87
9.2 Dénotation d'un composant de tableau 88
9.3 Modification sélective 89
9.4 Opérations sur les tableaux 89
9.5 Les tableaux en Java 89
9.6 Un exemple 91
9.7 Les chaînes de caractères 93
9.8 Exercices 94

CHAPITRE 10 • **L'ÉNONCÉ ITÉRATIF POUR** 97
10.1 Forme générale 97
10.2 Forme restreinte 98
10.3 L'énoncé `pour` de Java 98
10.4 Exemples 99
10.4.1 Le schéma de HORNER 99
10.4.2 Un tri interne simple 100
10.4.3 Confrontation de modèle 102
10.5 Complexité des algorithmes 105
10.6 Exercices 107

CHAPITRE 11 • **LES TABLEAUX À PLUSIEURS DIMENSIONS** 111
11.1 Déclaration 111
11.2 Dénotation d'un composant de tableau 112
11.3 Modification sélective 112
11.4 Opérations 113
11.5 Tableaux à plusieurs dimensions en Java 113
11.6 Exemples 113
11.6.1 Initialisation d'une matrice 113
11.6.2 Matrice symétrique 114
11.6.3 Produit de matrices 115
11.6.4 Carré magique 116
11.7 Exercices 118

CHAPITRE 12 • **HÉRITAGE** 123
12.1 Classes héritières 123
12.2 Redéfinition de méthodes 126
12.3 Recherche d'un attribut ou d'une méthode 127
12.4 Polymorphisme et liaison dynamique 127
12.5 Classes abstraites 129
12.6 Héritage simple et multiple 130
12.7 Héritage et assertions 130
12.7.1 Assertions sur les classes héritières 131
12.7.2 Assertions sur les méthodes 131
12.8 Relation d'héritage ou de clientèle 131
12.9 L'héritage en Java 132

CHAPITRE 13 • **LES EXCEPTIONS** 135
13.1 Émission d'une exception 135
13.2 Traitement d'une exception 136
13.3 Le mécanisme d'exception de Java 137
13.3.1 Traitement d'une exception 137
13.3.2 Émission d'une exception 138
13.4 Exercices 139

CHAPITRE 14 • **LES FICHIERS SÉQUENTIELS** 141
14.1 Déclaration de type 142
14.2 Notation 142
14.3 Manipulation des fichiers 143
14.3.1 Écriture 143
14.3.2 Lecture 144
14.4 Les fichiers de Java 145
14.4.1 Fichiers d'octets 145
14.4.2 Fichiers d'objets élémentaires 146

14.4.3 Fichiers d'objets structurés 150
14.5 Les fichiers de texte 151
14.6 Les fichiers de texte en Java 152
14.7 Exercices 155

CHAPITRE 15 • **RÉCURSIVITÉ** 157
15.1 Récursivité des actions 158
15.1.1 Définition 158
15.1.2 Finitude 158
15.1.3 Écriture récursive des sous-programmes 158
15.1.4 La pile d'évalution 161
15.1.5 Quand ne pas utiliser la récursivité? 162
15.1.6 Récursivité directe et croisée 164
15.2 Récursivité des objets 166
15.3 Exercices 169

CHAPITRE 16 • **STRUCTURES DE DONNÉES** 171
16.1 Définition d'un type abstrait 172
16.2 L'implantation d'un type abstrait 174
16.3 Utilisation du type abstrait 176
16.4 Généricité 176

CHAPITRE 17 • **STRUCTURES LINÉAIRES** 179
17.1 Les listes 179
17.1.1 Définition abstraite 180
17.1.2 L'implantation en Java 181
17.1.3 Énumération 192
17.2 Les piles 195
17.2.1 Définition abstraite 196
17.2.2 L'implantation en Java 196
17.3 Les files 199
17.3.1 Définition abstraite 199
17.3.2 L'implantation en Java 200
17.4 Les dèques 201
17.4.1 Définition abstraite 202
17.4.2 L'implantation en Java 202
17.5 Exercices 203

CHAPITRE 18 • **GRAPHES** 207
18.1 Terminologie 208
18.2 Définition abstraite d'un graphe 209
18.3 L'implantation en Java 211
18.3.1 Matrice d'adjacence 211

18.3.2 Listes d'adjacence 214
18.4 Parcours d'un graphe 215
18.4.1 Parcours en profondeur 215
18.4.2 Parcours en largeur 216
18.4.3 Programmation en Java des parcours de graphe 217
18.5 Exercices 219

CHAPITRE 19 • **STRUCTURES ARBORESCENTES** 221
19.1 Terminologie 222
19.2 Les arbres 223
19.2.1 Définition abstraite 224
19.2.2 L'implantation en Java 225
19.2.3 Algorithmes de parcours d'un arbre 228
19.3 Arbre binaire 229
19.3.1 Définition abstraite 230
19.3.2 L'implantation en Java 231
19.3.3 Parcours d'un arbre binaire 234
19.4 Représentation binaire des arbres généraux 235
19.5 Exercices 237

CHAPITRE 20 • **TABLES** 239
20.1 Définition abstraite 240
20.1.1 Ensembles 240
20.1.2 Description fonctionnelle 240
20.1.3 Description axiomatique 240
20.2 Représentation des éléments en Java 241
20.3 Représentation par une liste 242
20.3.1 Liste non ordonnée 242
20.3.2 Liste ordonnée 244
20.3.3 Recherche dichotomique 246
20.4 Représentation par un arbre ordonné 248
20.4.1 Recherche d'un élément 249
20.4.2 Ajout d'un élément 249
20.4.3 Suppression d'un élément 250
20.5 Les arbres AVL 252
20.5.1 Rotations 253
20.5.2 Mise en œuvre 256
20.6 Arbres 2-3-4 et bicolores 260
20.6.1 Les arbres 2-3-4 260
20.6.2 Mise en œuvre en Java 263
20.6.3 Les arbres bicolores 265
20.6.4 Mise en œuvre en Java 269
20.7 Tables d'adressage dispersé 275

20.7.1 Le problème des collisions 276
20.7.2 Choix de la fonction d'adressage 277
20.7.3 Résolution des collisions 278
20.8 Exercices 282

CHAPITRE 21 • **FILES AVEC PRIORITÉ** 285
21.1 Définition abstraite 285
21.2 Représentation avec une liste 286
21.3 Représentation avec un tas 286
21.3.1 Premier 287
21.3.2 Ajouter 288
21.3.3 Supprimer 289
21.3.4 L'implantation en Java 290
21.4 Exercices 293

CHAPITRE 22 • **ALGORITHMES DE TRI** 295
22.1 Introduction 295
22.2 Tris internes 296
22.2.1 L'implantation en Java 296
22.2.2 Méthodes par sélection 297
22.2.3 Méthodes par insertion 302
22.2.4 Tri par échanges 307
22.2.5 Comparaisons des méthodes 311
22.3 Tris externes 313
22.4 Exercices 316

CHAPITRE 23 • **ALGORITHMES SUR LES GRAPHES** 319
23.1 Composantes connexes 319
23.2 Fermeture transitive 321
23.3 Plus court chemin 324
23.3.1 L'implantation en Java 326
23.3.2 Complexité de l'algorithme 328
23.4 Tri topologique 328
23.4.1 L'implantation en Java 330
23.4.2 Existence de cycle dans un graphe 331
23.4.3 Tri topologique inverse 331
23.4.4 L'implantation en Java 332
23.5 Exercices 332

CHAPITRE 24 • **ALGORITHMES DE RÉTRO-PARCOURS** 335
24.1 Écriture récursive 335
24.2 Le problème des huit reines 337
24.3 Écriture itérative 339

24.4 Problème des sous-suites 340
24.5 Jeux de stratégie 342
24.5.1 Stratégie *MinMax* 342
24.5.2 Coupure α-β 346
24.5.3 Profondeur de l'arbre de jeu 348
24.6 Exercices 349

CHAPITRE 25 • **INTERFACES GRAPHIQUES** 353
25.1 Systèmes interactifs 353
25.2 Conception d'une application interactive 355
25.3 Environnements graphiques 357
25.3.1 Système de fenêtrage 357
25.3.2 Caractéristiques des fenêtres 360
25.3.3 Boîtes à outils 362
25.3.4 Générateurs 364
25.4 Interfaces graphiques en Java 364
25.4.1 Une simple fenêtre 365
25.4.2 Convertisseur d'euros 367
25.4.3 Un composant graphique pour visualiser des couleurs 370
25.4.4 Applets 373
25.5 Exercices 375

BIBLIOGRAPHIE 377

INDEX 381

Avant-propos

La seconde édition de cet ouvrage m'a donné l'occasion de compléter un de ses chapitres, d'en ajouter un autre et de corriger des erreurs. Une section sur les arbres équilibrés, qui manquait au chapitre 20 consacré aux tables, a été introduite. Elle traite les arbres AVL, 2-3-4 et bicolores et présente les algorithmes d'ajout et de suppression, ainsi que leur programmation en JAVA (rarement publiée dans la littérature). Par ailleurs, certains lecteurs regrettaient l'absence d'un chapitre sur les interfaces graphiques. Le chapitre 25 comble cette lacune. Enfin, de nombreuses, mais légères, retouches (corrections de coquilles, clarification du texte, etc.) ont été également apportées à ce livre.

Sophia Antipolis, mars 2004.

L'informatique est une science mais aussi une technologie et un ensemble d'outils. Ces trois composantes ne doivent pas être confondues, et l'enseignement de l'informatique ne doit pas être réduit au seul apprentissage des logiciels. Ainsi, l'activité de programmation ne doit pas se confondre avec l'étude d'un langage de programmation particulier. Même si l'importance de ce dernier ne doit pas être sous-estimée, il demeure un simple outil de mise en œuvre de concepts algorithmiques et de programmation généraux et fondamentaux. L'objectif de cet ouvrage est d'enseigner au lecteur des méthodes et des outils de construction de programmes informatiques valides et fiables.

L'étude de l'algorithmique et de la programmation est un des piliers fondamentaux sur lesquels repose l'enseignement de l'informatique. Ce livre s'adresse principalement aux étudiants des cycles informatiques et élèves ingénieurs informaticiens, mais aussi à tous ceux qui ne se destinent pas à la carrière informatique mais qui seront certainement confrontés au développement de programmes informatiques au cours de leur scolarité ou dans leur vie professionnelle.

Ce livre correspond au cours d'algorithmique et de programmation qui s'étend sur les deux premières années du premier cycle d'ingénieur de l'ESINSA, École Supérieure d'Ingénieurs de l'université de Nice-Sophia Antipolis.

Les quinze premiers chapitres sont le cours de première année. Ils présentent les concepts de base de la programmation impérative, en s'appuyant sur une méthodologie objet. Ils mettent en particulier l'accent sur la notion de preuve des programmes grâce à la notion d'affirmations (antécédent, conséquent, invariant) dont la vérification formelle garantit la validité de programmes, et introduisent la notion de complexité des algorithmes pour évaluer leur performance.

Les neuf derniers chapitres correspondent au cours de deuxième année. Ils étudient en détail les structures de données abstraites classiques et un certain nombre d'algorithmes fondamentaux que tout étudiant en informatique doit connaître et maîtriser.

La présentation des concepts de programmation cherche à être indépendante, autant que faire se peut, d'un langage de programmation particulier. Les algorithmes seront décrits dans une notation algorithmique épurée. Pour des raisons pédagogiques, il a toutefois bien fallu faire le choix d'un langage pour programmer les structures de données et les algorithmes présentés dans cet ouvrage. Ce choix s'est porté sur le langage à objets JAVA [GJS96], non pas par effet de mode, mais plutôt pour les qualités de ce langage, malgré quelques défauts. Ses qualités sont en particulier sa relative simplicité pour la mise en œuvre des algorithmes, un large champ d'application et sa grande disponibilité sur des environnements variés. Ce dernier point est en effet important ; le lecteur doit pouvoir disposer facilement d'un compilateur et d'un interprète afin de résoudre les exercices proposés à la fin des chapitres. Pour les défauts, on peut par exemple regretter l'absence de l'héritage multiple et de la généricité, et la présence de constructions archaïques héritées du langage C. Ce livre n'est pas un ouvrage d'apprentissage du langage JAVA. Même si les éléments du langage nécessaires à la mise en œuvre des notions d'algorithmique et de programmation ont été introduits, ce livre n'enseignera pas au lecteur les finesses et les arcanes de JAVA, pas plus qu'il ne décrira les nombreuses classes de l'API. Le lecteur intéressé pourra se reporter aux très nombreux ouvrages qui décrivent le langage en détail, comme par exemple [Bro99, Ska00, Eck00].

Les corrigés de la plupart des exercices, ainsi que des applets qui proposent une vision graphique de certains programmes présentés dans l'ouvrage sont accessibles sur le site Web de l'auteur à l'adresse :

`www.esinsa.unice.fr/~vg/Enseignement/AlgoProg`

Ce livre doit beaucoup à de nombreuses personnes. Tout d'abord, aux auteurs des algorithmes et des techniques de programmation qu'il présente. Il n'est pas possible de les citer tous ici, mais les références à leurs principaux textes sont dans la bibliographie. À Olivier Lecarme et Jean-Claude Boussard, mes professeurs à l'université de Nice qui m'ont enseigné cette discipline au début des années 80. Je tiens tout particulièrement à remercier ce dernier qui fut le tout premier lecteur attentif de cet ouvrage alors qu'il n'était encore qu'une ébauche, et qui m'a encouragé à poursuivre sa rédaction. À Johan Montagnat, qui est à l'origine de plusieurs exercices de fin de chapitre. À Carine Fédèle qui a bien voulu lire et corriger ce texte à de nombreuses reprises ; qu'elle en soit spécialement remercier. Enfin, à mes collègues et mes étudiants qui m'ont aidé et soutenu dans cette tâche ardue qu'est la rédaction d'un livre.

Sophia Antipolis, juin 2000.

Chapitre 1

Introduction

Les informaticiens, ou les simples usagers de l'outil informatique, utilisent des systèmes informatiques pour concevoir ou exécuter des programmes d'application. Nous considérerons qu'un environnement informatique est formé d'une part d'un ordinateur et de ses équipements externes, que nous appellerons l'environnement matériel, et d'autre part d'un système d'exploitation avec ses programmes d'application, que nous appellerons l'environnement logiciel. Les programmes qui forment le logiciel réclament des méthodes pour les construire et des langages pour les rédiger et les exécuter sur un ordinateur.

Dans ce chapitre, nous présenterons les notions de base de la programmation que sont l'environnement de développement et d'exécution d'un programme, les langages de programmation et les méthodes de construction des programmes.

1.1 ENVIRONNEMENT MATÉRIEL

Un *automate* est un ensemble fini de composants physiques pouvant prendre des états identifiables et reproductibles en nombre fini, auquel est associé un ensemble de changements d'états non instantanés qu'il est possible de commander et d'enchaîner sans intervention humaine.

Un *ordinateur* est un automate *déterministe* à composants électroniques. Tous les ordinateurs[1] actuels sont construits, peu ou prou, sur le modèle du mathématicien américain d'origine hongroise VON NEUMANN proposé en 1944. Un ordinateur est muni :

- D'une *mémoire* (dite *centrale* ou *principale*) qui contient deux sortes d'informations : d'une part l'information traitante, les *instructions*, et d'autre part l'information traitée,

1. Du moins, les ordinateurs monoprocesseur.

les *données*. Cette mémoire est formée d'un ensemble de cellules, ou *mots*, ayant chacun une adresse unique, et contenant des instructions ou des données. La représentation de l'information est faite grâce à une codification *binaire*, 0 ou 1. On appelle *longueur de mot*, caractéristique d'un ordinateur, le nombre d'éléments binaires, appelés *bits*, groupés dans une simple cellule. Les longueurs de mots usuelles des ordinateurs actuels (ou passés) sont, par exemple, 8, 16, 24, 32, 48 ou 64 bits. Cette mémoire possède une capacité *finie*, souvent exprimée en méga-octets (Mo) ; un *octet* est un ensemble de 8 bits, un kilo-octet (Ko) est égal à 1024 octets, un méga-octet est égal à 1024 Ko, et enfin un giga-octet (Go) est égal à 1024 Mo. Actuellement, les tailles courantes des mémoires centrales des ordinateurs varient entre 256 Mo à 512 Mo.

- D'une *unité centrale de traitement*, formée d'une *unité de commande* (UC) et d'une *unité arithmétique et logique* (UAL). L'unité de commande extrait de la mémoire centrale les instructions et les données sur lesquelles portent les instructions ; elle déclenche le traitement de ces données dans l'unité arithmétique et logique, et éventuellement range le résultat en mémoire centrale. L'unité arithmétique et logique effectue sur les données qu'elle reçoit les traitements commandés par l'unité de commande.
- De *registres*. Les registres sont des unités de mémorisation, en petit nombre (certains ordinateurs n'en ont pas), qui permettent à l'unité centrale de traitement de ranger de façon temporaire des données pendant les calculs. L'accès à ces registres est très rapide, beaucoup plus rapide que l'accès à une cellule de la mémoire centrale. Le rapport entre les temps d'accès à un registre et à la mémoire centrale est de l'ordre de 100.
- D'*unité d'échanges* reliées à des périphériques pour échanger de l'information avec le monde extérieur. L'unité de commande dirige les unités d'échange lorsqu'elle rencontre des instructions d'entrée-sortie.

Les ordinateurs actuels possèdent plusieurs niveaux de mémoire. Ils introduisent, entre le processeur et la mémoire centrale, des mémoires dites *caches* qui accélèrent l'accès aux données. Les mémoires caches peuvent être *primaires*, c'est-à-dire situées directement sur le processeur, ou *secondaires*, c'est-à-dire situées sur la carte mère. Le rapport entre le temps d'accès entre ces deux mémoires caches est d'environ 10. Le rapport entre le temps d'accès entre la mémoire cache secondaire et la mémoire centrale est lui aussi d'environ 10.

Les *équipements externes*, ou *périphériques*, sont un ensemble de composants permettant de relier l'ordinateur au monde extérieur, et notamment à ses utilisateurs humains. On peut distinguer :

- Les dispositifs qui servent pour la communication avec l'homme (clavier, écran, imprimantes, micros, haut-parleurs, scanners, etc.) et qui demandent une transcodification appropriée de l'information, par exemple sous forme de caractères alphanumériques.
- Les mémoires *secondaires* qui permettent de conserver de l'information, impossible à garder dans la mémoire centrale de l'ordinateur faute de place, ou que l'on désire conserver pour une période plus ou moins longue après l'arrêt de l'ordinateur. Les mémoires secondaires usuelles sont les disquettes, les disques, les CD-ROM, les bandes magnétiques (de moins en moins), etc. Notez que les disquettes sont des supports assez fragiles et donc peu fiables. Aujourd'hui, la capacité de certains disques approchent les 250 Go. L'accès aux informations sur les disques est plus lent que celui aux informations placées en mémoire centrale. Le rapport est d'environ 10.
- Les dispositifs qui permettent l'échange d'informations sur un réseau.

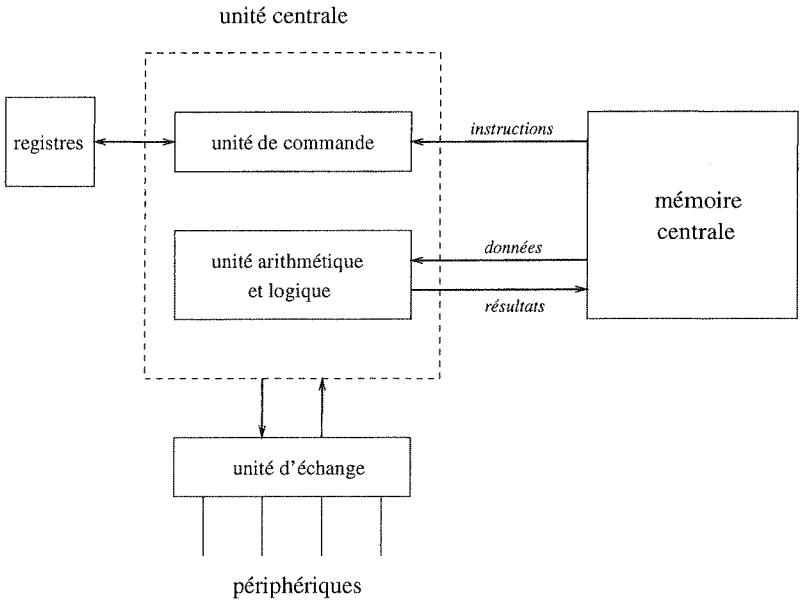


Figure 1.1 – *Structure générale d'un ordinateur.*

1.2 ENVIRONNEMENT LOGICIEL

L'ordinateur que fabrique le constructeur est une machine incomplète, à laquelle il faut ajouter, pour la rendre utilisable, une quantité importante de programmes variés, qui constitue le *logiciel.*

En général, un ordinateur est livré avec un *système d'exploitation.* Un système d'exploitation est un programme, ou plutôt un ensemble de programmes, qui assure la gestion des ressources, matérielles et logicielles, employées par le ou les utilisateurs. Un système d'exploitation a pour tâche la gestion et la conservation de l'information (gestion des processus et de la mémoire centrale, système de gestion de fichiers) ; il a pour rôle de créer l'environnement nécessaire à l'exécution d'un travail, et est chargé de répartir les ressources entre les usagers. Entre l'utilisateur et l'ordinateur, il propose une interface *textuelle* au moyen d'un *interprète de commandes* et une interface *graphique* au moyen d'un *gestionnaire de fenêtres.*

Les systèmes d'exploitation des premiers ordinateurs ne permettaient l'exécution que d'une seule tâche à la fois, selon un mode de fonctionnement appelé traitement pas lots. À partir des années 60, les systèmes d'exploitation ont cherché à exploiter au mieux les ressources des ordinateurs en permettant le *temps partagé*, pour offrir un accès simultané à plusieurs utilisateurs.

Jusqu'au début des années 80, les systèmes d'exploitation étaient dits « propriétaires ». Les constructeurs fournissaient avec leurs machines un système d'exploitation spécifique et le nombre de systèmes d'exploitation différents était important. Aujourd'hui, ce nombre a considérablement réduit, et seuls quelques-uns sont réellement utilisés dans le monde. Citons, par exemple, WINDOWS et MACOS pour les ordinateurs individuels, et UNIX pour les ordinateurs multi-utilisateurs.

Aujourd'hui avec l'augmentation de la puissance des ordinateurs personnels, et l'avènement des réseaux mondiaux, les systèmes d'exploitation offrent, en plus des fonctions déjà citées, une quantité extraordinaire de services et d'outils aux utilisateurs. Ces systèmes d'exploitation modernes mettent à la disposition des utilisateurs tout un ensemble d'applications (traitement de texte, tableurs, outils multimédia, navigateurs pour le WEB, ...) qui leur offre un environnement de travail pré-construit, confortable et facile d'utilisation.

Le traitement de l'information est l'exécution par l'ordinateur d'une série finie de commandes préparées à l'avance, le *programme*, et qui vise à calculer et rendre des résultats, généralement, en fonction de données entrées au début ou en cours d'exécution. Les commandes qui forment le programme sont décrites au moyen d'un *langage*. Si ces commandes se suivent strictement dans le temps, et ne s'exécutent jamais simultanément, l'exécution est dite *séquentielle*, sinon elle est dite *parallèle*.

Chaque ordinateur possède un langage qui lui est propre, appelé *langage machine*. Le langage machine est un ensemble de commandes élémentaires représentées en code binaire qu'il est possible de faire exécuter par l'unité centrale de traitement d'un ordinateur donné. Le seul langage que comprend l'ordinateur est son langage machine.

Tout logiciel est écrit à l'aide d'un ou plusieurs *langages de programmation*. Un langage de programmation est un ensemble d'énoncés déterministes, qu'il est possible, pour un être humain, de rédiger selon les règles d'une grammaire donnée et destinés à représenter les objets et les commandes pouvant entrer dans la constitution d'un programme. Ni le langage machine, trop éloigné des modes d'expressions humains, ni les langues naturelles écrites ou parlées, trop ambiguës, ne sont des langages de programmation.

La production de logiciel est une activité difficile et complexe et les éditeurs font payer, parfois très cher, leur logiciel dont le code source n'est, en général, pas distribué. Toutefois, tous les logiciels ne sont pas payants. La communauté internationale des informaticiens produit depuis longtemps du logiciel gratuit (ce qui ne veut pas dire qu'il est de mauvaise qualité, bien au contraire) mis à la disposition de tous. Il existe aux États-Unis, une fondation, la FSF (*Free Software Foundation*), à l'initiative de R. STALLMAN, qui a pour but la promotion de la construction du logiciel *libre*, ainsi que celle de sa distribution. Libre ne veut pas dire nécessairement gratuit[2], bien que cela soit souvent le cas, mais indique que le texte source du logiciel est disponible.

1.3 LES LANGAGES DE PROGRAMMATION

Nous venons de voir que chaque ordinateur possède un langage qui lui est propre : le langage machine, qui est en général totalement incompatible avec celui d'un ordinateur d'un autre modèle. Ainsi, un programme écrit dans le langage d'un ordinateur donné ne pourra être réutilisé sur un autre ordinateur.

Le langage *d'assemblage* est un codage alphanumérique du langage machine. Il est plus lisible que ce dernier et surtout permet un adressage relatif de la mémoire. Toutefois, comme le langage machine, le langage d'assemblage est lui aussi dépendant d'un ordinateur donné (voire d'une famille d'ordinateurs) et ne facilite pas le transport des programmes vers des

2. La confusion provient du fait qu'en anglais le mot « free » possède les deux sens.

machines dont l'architecture est différente. L'exécution d'un programme écrit en langage d'assemblage nécessite sa traduction préalable en langage machine par un programme spécial, l'*assembleur*.

Le texte qui suit, écrit en langage d'assemblage d'un PENTIUM II, correspond à l'expression $sin(2) + 1$.

```
pushl $1073741824
pushl $0
call sin
addl $8,%esp
fldl
faddp %st,%st(1)
```

Le langage d'assemblage, comme le langage machine, est d'un niveau très élémentaire (une suite linéaire de commandes et sans structure) et, comme le montre l'exemple précédent, guère lisible et compréhensible. Son utilisation par un être humain est alors difficile, fastidieuse et sujette à erreurs.

Ces défauts, entre autres, ont conduit à la conception des langages de programmation dits de *haut niveau*. Un langage de programmation de haut niveau offrira au programmeur des moyens d'expression structurés proches des problèmes à résoudre et qui amélioreront la fiabilité des programmes. Pendant de nombreuses années, les ardents défenseurs de la programmation en langage d'assemblage avançaient le critère de son efficacité. Les optimiseurs de code ont balayé cet argument depuis longtemps, et les défauts de ces langages sont tels que leurs thuriféraires sont de plus en plus rares.

Si on ajoute à un ordinateur un langage de programmation, tout se passe comme si l'on disposait d'un nouvel ordinateur (une machine abstraite), dont le langage est maintenant adapté à l'être humain, aux problèmes qu'il veut résoudre et à la façon qu'il a de comprendre et de raisonner. De plus, cet ordinateur fictif pourra recouvrir des ordinateurs différents, si le langage de programmation peut être installé sur chacun d'eux. Ce dernier point est très important, puisqu'il signifie qu'un programme écrit dans un langage de haut niveau pourra être exploité (théoriquement) sans modification sur des ordinateurs différents.

La définition d'un langage de programmation recouvre trois aspects fondamentaux. Le premier, appelé *lexical*, définit les symboles (ou caractères) qui servent à la rédaction des programmes et les règles de formation des mots du langage. Par exemple, un entier décimal sera défini comme une suite de chiffres compris entre 0 et 9. Le second, appelé *syntaxique*, est l'ensemble des règles grammaticales qui organisent les mots en phrases. Par exemple, la phrase « 234 / 54 », formée de deux entiers et d'un opérateur de division, suit la règle grammaticale qui décrit une expression. Le dernier aspect, appelé *sémantique*, étudie la signification des phrases. Il définit les règles qui donnent un sens aux phrases. Notez qu'une phrase peut être syntaxiquement valide, mais incorrecte du point de vue de sa sémantique (*e.g.* 234/0, une division par zéro est invalide). L'ensemble des règles lexicales, syntaxiques et sémantiques définit un langage de programmation, et on dira qu'un programme appartient à un langage de programmation donné s'il vérifie cet ensemble de règles. La vérification de la conformité lexicale, syntaxique et sémantique d'un programme est assurée automatiquement

par des analyseurs qui s'appuient, en général, sur des notations formelles qui décrivent sans ambiguïté l'ensemble des règles.

Comment exécuter un programme rédigé dans un langage de programmation de haut niveau sur un ordinateur qui, nous le savons, ne sait traiter que des programmes écrits dans son langage machine ? Voici deux grandes familles de méthodes qui permettent de résoudre ce problème :

- La première méthode consiste à *traduire* le programme, appelé *source*, écrit dans le langage de haut niveau, en un programme sémantiquement *équivalent* écrit dans le langage machine de l'ordinateur (voir la figure 1.2). Cette traduction est faite au moyen d'un logiciel spécialisé appelé *compilateur*. Un compilateur possède au moins quatre phases : trois phases d'analyse (lexicale, syntaxique et sémantique), et une phase de production de code machine. Bien sûr, le compilateur ne produit le code machine que si le programme source respecte les règles du langage, sinon il devra signaler les erreurs au moyen de messages précis. En général, le compilateur produit du code pour un seul type de machine, celui sur lequel il est installé. Notez que certains compilateurs, dits *multicibles*, produisent du code pour différentes familles d'ordinateurs.

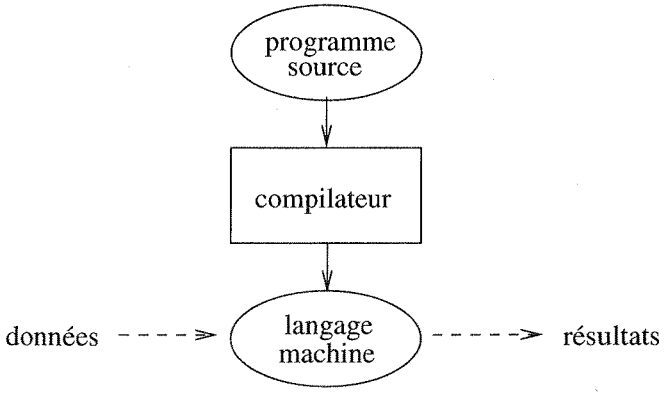


Figure 1.2 – *Traduction en langage machine.*

- Nous avons vu qu'un langage de programmation définit un ordinateur fictif. La seconde méthode consiste à *simuler* le fonctionnement de cet ordinateur fictif sur l'ordinateur réel par *interprétation* des instructions du langage de programmation de haut niveau. Le logiciel qui effectue cette interprétation s'appelle un *interprète*. L'interprétation directe des instructions du langage est en général difficilement réalisable. Une première phase de traduction du langage de haut niveau vers un langage *intermédiaire* de plus bas niveau est d'abord effectuée. Remarquez que cette phase de traduction comporte les mêmes phases d'analyse qu'un compilateur. L'interprétation est alors faite sur le langage intermédiaire. C'est la technique d'implantation du langage JAVA (voir la figure 1.3), mais aussi de beaucoup d'autres langages. Un programme source JAVA est d'abord traduit en un programme objet écrit dans un langage intermédiaire, appelé JAVA pseudo-code (ou *byte-code*). Le programme objet est ensuite exécuté par la machine virtuelle JAVA (JVM).

Ces deux méthodes, compilation et interprétation, ne sont pas incompatibles, et bien souvent pour un même langage les deux techniques sont mises en œuvre. L'intérêt de l'interpréta-

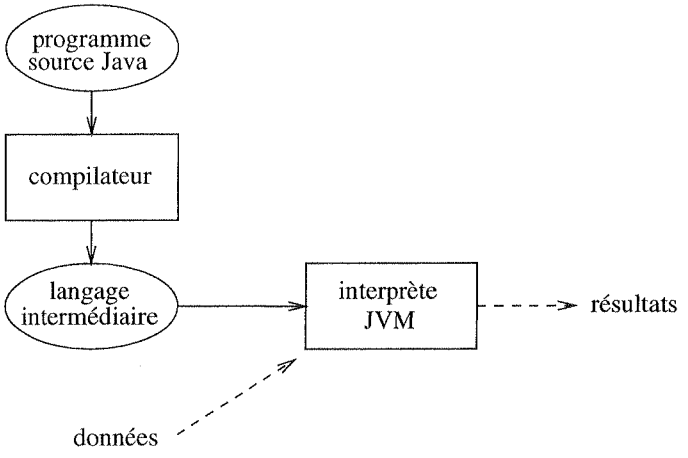


Figure 1.3 – *Traduction et interprétation d'un programme* JAVA.

tion est d'assurer au langage, ainsi qu'aux programmes, une grande *portabilité*. Ils dépendent faiblement de leur environnement d'implantation et peu ou pas de modifications sont nécessaires à leur exécution dans un environnement différent. Son inconvénient majeur est que le temps d'exécution des programmes interprétés est notablement plus important que celui des programmes compilés.

Bref historique

La conception des langages de programmation a souvent été influencée par un domaine d'application particulier, un type d'ordinateur disponible, ou les deux à la fois. Depuis près de cinquante ans, plusieurs centaines de langages de programmation ont été conçus. Certains n'existent plus, d'autres ont eu un usage limité, et seule une minorité sont vraiment très utilisés. Le but de ce paragraphe est de donner quelques repères importants dans l'histoire des langages de programmation « classiques ». Il n'est pas question de dresser ici un historique exhaustif. Le lecteur pourra se reporter avec intérêt aux ouvrages [Sam69, Wex81, Hor83] qui retracent les vingt-cinq premières années de cette histoire, à [ACM93] pour les quinze années qui suivirent, et à [M+89] qui présente un panorama complet des langages à objets.

FORTRAN (*Formula Translator*) [Int57, ANS78] fut le premier traducteur en langage machine d'une notation algébrique pour écrire des formules mathématiques. Il fut conçu à IBM à partir de 1954 par J. BACKUS en collaboration avec d'autres chercheurs. Jusqu'à cette date, les programmes étaient écrits en langage machine ou d'assemblage, et l'importance de FORTRAN a été de faire la démonstration, face au scepticisme de certains, de l'efficacité de la traduction automatique d'une notation évoluée pour la rédaction de programmes de calcul numérique scientifique. À l'origine, FORTRAN n'est pas un langage et ses auteurs n'en imaginaient pas la conception. En revanche, il ont inventé des techniques d'optimisation de code particulièrement efficaces.

LISP (*List Processor*) a été développé à partir de la fin de l'année 1958 par J. MCCARTHY au MIT pour le traitement de données symboliques (*i.e.* non numériques) dans le do-

maine de l'intelligence artificielle. Il fut utilisé pour résoudre des problèmes d'intégration et de différenciation symboliques, de preuve de théorèmes, ou encore de géométrie et a servi au développement de modèles théoriques de l'informatique. La notation utilisée, appelée *S-expression*, est plus proche d'un langage d'assemblage d'une machine abstraite spécialisée dans la manipulation de liste (le type de donnée fondamental ; un programme LISP est lui-même une liste), que d'un véritable langage de programmation. Cette notation préfixée de type fonctionnel utilise les expressions conditionnelles et les fonctions récursives basées sur la notation λ (*lambda*) de A. CHURCH. Une notation, appelée *M-expression*, s'inspirant de FORTRAN et à traduire en *S-expression*, avait été conçue à la fin des années 50 par J. MCCARTHY, mais n'a jamais été implémentée. Hormis LISP 2, un langage inspiré de ALGOL 60 (voir paragraphes suivants) développé et implémenté au milieu des années 60, les nombreuses versions et variantes ultérieures de LISP seront basées sur la notation *S-expression*. Il est à noter aussi que LISP est le premier à mettre en œuvre un système de récupération automatique de mémoire (*garbage-collector*).

La gestion est un autre domaine important de l'informatique. Au cours des années 50 furent développer plusieurs langages de programmation spécialisés dans ce domaine. À partir de 1959, un groupe de travail comprenant des universitaires, mais surtout des industriels américains, sous l'égide du Département de la Défense des États-Unis (DOD), réfléchit à la conception d'un langage d'applications de gestion commun. Le langage COBOL (*Common Business Oriented Language*) [Cob60] est le fruit de cette réflexion. Il a posé les premières bases de la structuration des données.

On peut dire que les années 50 correspondent à l'approche expérimentale de l'étude des concepts des langages de programmation. Il est notable que FORTRAN, LISP et COBOL, sous des formes qui ont bien évolué, sont encore largement utilisés aujourd'hui. Les années 60 correspondent à l'approche mathématique de ces concepts, et le développement de ce qu'on appelle la théorie des langages. En particulier, beaucoup de notations formelles sont apparues pour décrire la sémantique des langages de programmation.

De tous les langages, ALGOL 60 (*Algorithmic Language*) [Nau60] est celui qui a eu le plus d'influence sur les autres. C'est le premier langage défini par un comité international (présidé par J. BACKUS et presque uniquement composé d'universitaires), le premier à séparer les aspects lexicaux et syntaxiques, à donner une définition syntaxique formelle (la Forme Normale de BACKUS, BNF), et le premier à soumettre la définition à l'ensemble de la communauté pour en permettre la révision avant de figer quoi que ce soit. De nombreux concepts, que l'on retrouvera dans la plupart des langages de programmation qui suivront, ont été définis pour la première fois dans ALGOL 60 (la structure de bloc, le concept de déclaration, le passage des paramètres, les procédures récursives, les tableaux dynamiques, les énoncés conditionnels et itératifs, le modèle de pile d'exécution, *etc.*). Pour toutes ces raisons, et malgré quelques lacunes mises en évidence par D. KNUTH [Knu67], ALGOL 60 est le langage qui fit le plus progresser l'informatique.

Dans ces années 60, des tentatives de définition de langages *universels*, c'est-à-dire pouvant s'appliquer à tous les domaines, ont vu le jour. Les langages PL/I (*Programming Language One*) [ANS76] et ALGOL 68 reprennent toutes les « bonnes » caractéristiques de leurs aînés conçus dans les années 50. PL/I cherche à combiner en un seul langage COBOL, LISP, FORTRAN (entre autres langages), alors qu'ALGOL 68 est le successeur *officiel* d'ALGOL 60. Ces langages, de part la trop grande complexité de leur définition, et par conséquence de leur utilisation, n'ont pas connu le succès attendu.

Lui aussi fortement inspiré par ALGOL 60, PASCAL [NAJN75, AFN82] est conçu par N. WIRTH en 1969. D'une grande simplicité conceptuelle, ce langage algorithmique a servi (et peut-être encore aujourd'hui) pendant de nombreuses années à l'enseignement de la programmation dans les universités.

Le langage C [KR88, ANS89] a été développé en 1972 par D. RITCHIE pour la réécriture du système d'exploitation UNIX. Conçu à l'origine comme langage d'écriture de système, ce langage est utilisé pour la programmation de toutes sortes d'applications. Malgré de nombreux défauts, C est encore très utilisé aujourd'hui, sans doute pour des raisons d'efficacité du code produit et une certaine portabilité des programmes. Ce langage a été normalisé en 1989 par l'ANSI[3].

Les années 70 correspondent à l'approche « génie logiciel ». Devant le coût et la complexité toujours croissants des logiciels, il devient essentiel de développer de nouveaux langages puissants, ainsi qu'une méthodologie pour guider la construction, maîtriser la complexité, et assurer la fiabilité des programmes. ALPHARD [W+76] et CLU [L+77], deux langages expérimentaux, MODULA-2 [Wir85], ou encore ADA [ANS83] sont des exemples parmi d'autres de langages imposant une méthodologie dans la conception des programmes. Une des originalités du langage ADA est certainement son mode de définition. Il est le produit d'un appel d'offres international lancé en 1974 par le DOD pour unifier la programmation de ses systèmes embarqués. Suivirent de nombreuses années d'étude de conception pour déboucher sur une norme (ANSI, 1983), posée comme préalable à l'exploitation du langage.

Les langages des années 80-90, dans le domaine du génie logiciel, mettent en avant le concept de la *programmation objet*. Cette notion n'est pas nouvelle puisqu'elle date de la fin des années 60 avec SIMULA [DN66], certainement le premier langage à objets. Toutefois, ce n'est que récemment qu'elle connaît une certaine vogue. SMALLTALK [GR89], C++ [Str86] (issu de C), EIFFEL [Mey92], ou JAVA [GJS96] sont, parmi les très nombreux langages à objets, les plus connus. JAVA, apparu récemment, connaît un grand engouement, en particulier grâce au WEB d'Internet.

Dans le domaine, de l'intelligence artificielle, nous avons déjà cité LISP. Un autre langage, le langage *déclaratif* PROLOG (Programmation en Logique) [CKvC83], conçu dès 1972 par l'équipe marseillaise de A. COLMERAUER, a connu une grande notoriété dans les années 80. PROLOG est issu de travaux sur le dialogue homme-machine en langage naturel et sur les démonstrateurs automatiques de théorèmes. Un programme PROLOG ne s'appuie plus sur un algorithme, mais sur la déclaration d'un ensemble de règles à partir desquelles les résultats pourront être déduits par unification et rétro-parcours (*backtracking*) à l'aide d'un évaluateur spécialisé.

Nous finirons cet historique par le langage ICON [GHK79]. Il est le dernier d'une famille de langages de manipulation de chaînes de caractères (SNOBOL 1 à 4 et SL5) conçus par R. GRISWOLD dès 1960 pour le traitement de données symboliques. Ces langages intègrent le mécanisme de confrontation de modèles (*pattern matching*), la notion de succès et d'échec de l'évaluation d'une expression, mais l'idée la plus originale introduite par ICON est celle du mécanisme de générateur et d'évaluation dirigée par le but. Un générateur est une expression qui peut fournir zéro ou plusieurs résultats, et l'évaluation dirigée par le but permet d'exploiter les séquences de résultats produites par les générateurs. Ces langages ont connu un vif succès et il existe aujourd'hui encore une grande activité autour du langage ICON.

3. *American National Standards Institute*, l'institut de normalisation des États-Unis.

1.4 CONSTRUCTION DES PROGRAMMES

L'activité de programmation est difficile et complexe. Le but de tout programme est de calculer et retourner des résultats *valides* et *fiables*. Quelle que soit la taille des programmes, de quelques dizaines de lignes à plusieurs centaines de milliers, la conception des programmes exige des méthodes rigoureuses, si les objectifs de justesse et fiabilité veulent être atteints.

D'une façon très générale, on peut dire qu'un programme effectue des *actions* sur des *objets*. Jusque dans les années 60, la structuration des programmes n'était pas un souci majeur. C'est à partir des années 70, face à des coûts de développement des logiciels croissants, que l'intérêt pour la structuration des programmes s'est accrue. À cette époque, les méthodes de construction des programmes commençaient par structurer les actions. La structuration des objets venait ultérieurement. Depuis la fin des années 80, le processus est inversé. Essentiellement pour des raisons de pérennité (relative) des objets par rapport à celle des actions : les programmes sont structurés d'*abord* autour des objets. Les choix de structuration des actions sont fixés par la suite.

Lorsque le choix des actions précède celui des objets, le problème à résoudre est décomposé, en termes d'actions, en sous-problèmes plus simples, eux-mêmes décomposés en d'autres sous-problèmes encore plus simples, jusqu'à obtenir des éléments directement programmables. Avec cette méthode de construction, souvent appelée *programmation descendante par raffinements successifs*, la représentation particulière des objets, sur lesquels portent les actions, est retardée le plus possible. L'analyse du problème à traiter se fait dans le sens descendant d'une arborescence, dont chaque nœud correspond à un sous-problème bien déterminé du programme à construire. Au niveau de la racine de l'arbre, on trouve le problème posé dans sa forme initiale. Au niveau des feuilles, correspondent des actions pouvant s'énoncer directement et sans ambiguïté dans le langage de programmation choisi. Sur une même branche, le passage du nœud père à ses fils correspond à un accroissement du niveau de détail avec lequel est décrite la partie correspondante. Notez que sur le plan horizontal, les différents sous-problèmes doivent avoir chacun une cohérence propre et donc minimiser leur nombre de relations.

En revanche, lorsque le choix des objets précède celui des actions, la structure du programme est fondée sur les objets et sur leurs interactions. Le problème à résoudre est vu comme une modélisation (opérationnelle) d'un aspect du monde réel constitué d'objets. Cette vision est particulièrement évidente avec les logiciels graphiques et plus encore, de simulation. Les objets sont des composants qui contiennent des *attributs* (données) et des *méthodes* (actions) qui décrivent le comportement de l'objet. La communication entre objets se fait par *envoi de messages*, qui donne l'accès à un attribut ou qui lance une méthode.

Les critères de fiabilité et de validité ne sont pas les seuls à caractériser la qualité d'un programme. Il est fréquent qu'un programme soit modifié pour apporter de nouvelles fonctionnalités ou pour évoluer dans des environnements différents, ou soit dépecé pour fournir « des pièces détachées » à d'autres programmes. Ainsi de nouveaux critères de qualité, tels que l'*extensibilité*, la *compatibilité* ou la *réutilisabilité*, viennent s'ajouter aux précédents. Nous verrons que l'approche objet, bien plus que la méthode traditionnelle de décomposition fonctionnelle, permet de mieux respecter ces critères de qualité.

Les actions mises en jeu dans les deux méthodologies précédentes reposent sur la notion

d'*algorithme*[4]. L'algorithme décrit, de façon non ambiguë, l'ordonnancement des actions à effectuer dans le temps pour spécifier une fonctionnalité à traiter de façon automatique. Il est dénoté à l'aide d'une notation formelle, qui peut être indépendante du langage utilisé pour le programmer.

La conception d'algorithme est une tâche difficile qui nécessite une grande réflexion. Notez que le travail requis pour l'exprimer dans une notation particulière, c'est-à-dire la programmation de l'algorithme dans un langage particulier, est réduit par comparaison à celui de sa conception. *La réflexion sur papier, stylo en main, sera le préalable à toute programmation sur ordinateur.*

Pour un même problème, il existe bien souvent plusieurs algorithmes qui conduisent à sa solution. Le choix du « meilleur » algorithme est alors généralement guidé par des critères d'efficacité. La *complexité* d'un algorithme est une mesure théorique de ses performances en fonction d'éléments caractéristiques de l'algorithme. Le mot *théorique* signifie en particulier que la mesure est indépendante de l'environnement matériel et logiciel. Nous verrons à la section 10.5 page 105 comment établir cette mesure.

Le travail principal dans la conception d'un programme résidera dans le choix des objets qui le structureront, la validation de leurs interactions et le choix et la vérification des algorithmes sous-jacents.

1.5 DÉMONSTRATION DE VALIDITÉ

Notre but est de construire des programmes valides, c'est-à-dire conformes à ce que l'on attend d'eux. Comment vérifier la validité d'un programme ? Une fois le programme écrit, on peut, par exemple, tester son exécution. Si la phase de test, c'est-à-dire la vérification expérimentale par l'exécution du programme sur des données particulières, est nécessaire, elle ne permet en aucun cas de démontrer la justesse à 100% du programme. En effet, il faudrait faire un test *exhaustif* sur l'ensemble des valeurs possibles des données. Ainsi, pour une simple addition de deux entiers codés sur 32 bits, soit 2^{32} valeurs possibles par entier, il faudrait tester $2^{32\times 2}$ opérations. Pour une μ-seconde par opération, il faudrait 9×10^9 années ! N. WIRTH résume cette idée dans [Wir75] par la formule suivante :

> « *L'expérimentation des programmes peut servir à montrer la présence d'erreurs, mais jamais à prouver leur absence.* »

La preuve[5] de la validité d'un programme ne pourra donc se faire que *formellement* de façon *analytique*, tout le long de la construction du programme et, évidemment, pas une fois que celui-ci est terminé.

4. Le mot algorithme ne vient pas comme certains le pensent, du mot logarithme, mais doit son origine à un mathématicien persan du IX^e siècle, dont le nom abrégé était AL-KHOWÂRIZMÎ (de la ville de Khowârizm). Cette ville située dans l'Üzbekistān, s'appelle aujourd'hui Khiva. Notez toutefois que cette notion est bien plus ancienne. Les Babyloniens de l'Antiquité, les Égyptiens ou les Grecs avaient déjà formulé des règles pour résoudre des équations. Euclide (vers 300 av. J.C.) conçut un algorithme permettant de trouver le *pgcd* de deux nombres.

5. La preuve de programme est un domaine de recherche théorique ancien, mais toujours ouvert et très actif.

La technique que nous utiliserons est basée sur des *assertions* qui décriront les propriétés des éléments (objets, actions) du programme. Par exemple, une assertion indiquera qu'en tel point du programme telle valeur entière est négative.

Nous parlerons plus tard des assertions portant sur les objets. Celles pour décrire les propriétés des actions, c'est-à-dire leur sémantique, suivront l'*axiomatique* de C.A.R. HOARE [Hoa69]. L'assertion qui précède une action s'appelle l'*antécédent* ou *pré-condition* et celle qui la suit le *conséquent* ou *post-condition*.

Pour chaque action du programme, il sera possible, grâce à des *règles de déduction*, de déduire de façon *systématique* le conséquent à partir de l'antécédent. Notez qu'il est également possible de déduire l'antécédent à partir du conséquent. Ainsi pour une tâche particulière, formée par un enchaînement d'actions, nous pourrons démontrer son exactitude, c'est-à-dire le passage de l'antécédent initial jusqu'au conséquent final, par application des règles de déduction sur toutes les actions qui le composent.

Il est très important de comprendre, que les affirmations d'un programme ne doivent pas être définies *a posteriori*, c'est-à-dire une fois le programme écrit, mais bien au contraire *a priori* puisqu'il s'agit de construire l'action en fonction de l'effet prévu.

Une action `A` avec son *antécédent* et son *conséquent* sera dénotée :

```
{antécédent}
A
{conséquent}
```

Les assertions doivent être les plus formelles possibles, si l'on désire *prouver* la validité du programme. Elles s'apparentent d'ailleurs à la notion mathématique de *prédicat*. Toutefois, il sera nécessaire de trouver un compromis entre leur complexité et celle du programme. En d'autres termes, s'il est plus difficile de construire ces assertions que le programme lui-même, on peut se demander quel est leur intérêt ?

Certains langages de programmation, en fait un nombre réduit[6], intègrent des mécanismes de vérification de la validité des assertions spécifiées par les programmeurs. Dans ces langages, les assertions font donc parties intégrantes du programme. Elles sont contrôlées au fur et à mesure de l'exécution du programme, ce qui permet de détecter une situation d'erreur. Dans les autres langages, les assertions, même si elles ne sont pas traitées automatiquement par le système, devront être exprimées sous forme de *commentaires*. Ces commentaires serviront à l'auteur du programme, ou aux lecteurs, à se convaincre de la validité du programme.

6. Citons certains langages expérimentaux conçus dans les années 70, tels que ALPHARD [M. 81], ou plus récemment EIFFEL.

Chapitre 2

Actions élémentaires

Un programme est un processus de calcul qui peut être modélisé de différentes façons. Nous considérons tout d'abord qu'un programme est une suite de commandes qui effectuent des *actions* sur des données appelées *objets*, et qu'il peut être décrit par une fonction f dont l'ensemble de départ $\mathcal{D}$ est un ensemble de données, et l'ensemble d'arrivée $\mathcal{R}$ est un ensemble de résultats :

$$f : \mathcal{D} \rightarrow \mathcal{R}$$

À ce schéma, on peut faire correspondre trois premières actions *élémentaires*, ou *énoncés simples*, que sont la lecture d'une donnée, l'exécution d'une procédure ou d'une fonction prédéfinie sur cette donnée et l'écriture d'un résultat.

2.1 LECTURE D'UNE DONNÉE

La *lecture* d'une donnée consiste à faire entrer un objet en mémoire centrale à partir d'un équipement externe. Selon le cas, cette action peut préciser l'équipement sur lequel l'objet doit être lu, et où il se situe sur cet équipement. La façon d'exprimer l'ordre de lecture varie bien évidemment d'un langage à un autre. Pour l'instant, nous nous occuperons uniquement de lire des données au clavier, c'est-à-dire sur l'*entrée standard* et nous appellerons *lire* l'action de lecture.

Une fois lu, l'objet placé en mémoire doit porter un nom, permettant de le distinguer sans ambiguïté des objets déjà présents. Ce nom sera cité chaque fois qu'on utilisera l'objet en question dans la suite du programme. C'est l'action de lecture qui précise le nom de l'objet

lu. La lecture d'un objet sur l'entrée standard à placer en mémoire centrale sous le nom x s'écrit de la façon suivante :

```
{il existe une donnée à lire sur l'entrée standard}
lire(x)
{une donnée a été lue sur l'entrée standard,
 placée en mémoire centrale et le nom x permet de la désigner}
```

Notez que plusieurs commandes de lecture peuvent être exécutées les unes à la suite des autres. Si le même nom est utilisé chaque fois, il désignera la dernière donnée lue.

2.2 EXÉCUTION D'UNE PROCÉDURE PRÉDÉFINIE

L'objet qui vient d'être lu et placé en mémoire peut être la donnée d'un calcul, et en particulier la donnée d'une *procédure* ou d'une *fonction prédéfinie*. On dit alors que l'objet est un *paramètre effectif* « *donnée* » de la procédure ou de la fonction.

Ces procédures ou ces fonctions sont souvent conservées dans des bibliothèques et sont directement accessibles par le programme. Traditionnellement, les langages de programmation proposent des fonctions mathématiques et des procédures d'entrées-sorties.

L'exécution d'une procédure ou d'une fonction est une action élémentaire qui correspond à ce qu'on nomme un *appel* de procédure ou de fonction. Par exemple, la notation `sin(x)` est un appel de fonction qui calcule le sinus de `x`, `x` désignant le nom de l'objet en mémoire.

Une fois l'appel d'une procédure ou d'une fonction effectué, comment récupérer le résultat du calcul ?

S'il s'agit d'une fonction `f`, la notation `f(x)` sert à la fois à commander l'appel et à nommer le résultat. C'est la notion de fonction des mathématiciens. Ainsi, `sin(x)` est à la fois l'appel de la fonction et le résultat.

```
{le nom x désigne une valeur en mémoire}
sin(x)
{l'appel de sin(x) a calculé le sinus de x}
```

Bien évidemment, il est possible de fournir plusieurs paramètres « donnée » lors de l'appel d'une fonction. Par exemple, la notation `f(x,y,z)` correspond à l'appel d'une fonction f avec trois données nommées respectivement `x`, `y` et `z`.

Il peut être également utile de donner un nom au résultat. Avec une procédure, il sera possible de préciser ce nom au moment de l'appel, sous la forme d'un second paramètre, appelé paramètre effectif « *résultat* ».

```
{le nom x désigne une valeur en mémoire}
P(x,y)
{l'appel de la procédure P sur la donnée x}
{a calculé un résultat désigné par y}
```

Comme une fonction, une procédure peut posséder plusieurs paramètres « donnée ». De plus, si elle produit plusieurs résultats, ils sont désignés par plusieurs paramètres « résultat ».

```
{les noms x et y désignent des valeurs en mémoire}
P(x,y,a,b,c)
{l'appel de la procédure P sur les données x et y
 a calculé trois résultats désignés par a, b, c}
```

Remarquez que rien dans la notation de cet appel de procédure ne distinguent les paramètres effectifs « donnée » des paramètres effectifs « résultat ». Nous verrons au chapitre 6 comme s'opère cette distinction.

Notez également que puisqu'une fonction ne produit qu'un seul résultat donné par la dénotation de l'appel, une fonction ne doit pas posséder de paramètre « résultat ». Certains langages de programmation en font une règle, mais malheureusement, bien souvent, ils autorisent les fonctions à posséder des paramètres « résultat ».

2.3 ÉCRITURE D'UN RÉSULTAT

Une fois le résultat d'une procédure ou d'une fonction calculé, il est souvent souhaitable de récupérer ce résultat sur un équipement externe. Il existe pour cela une action élémentaire réciproque de celle de lecture. C'est l'action d'*écriture*. Elle consiste à transférer vers un équipement externe désigné, la valeur d'un objet en mémoire. Une transcodification est associée à cette action dans le cas où le destinataire final est un être humain.

Pour l'instant, nous écrirons les résultats sur l'écran, qu'on nomme la *sortie standard*.

```
{le nom y désigne une valeur en mémoire}
écrire(y)
{la valeur de y a été écrite sur la sortie standard}
```

Notez que les actions de lecture et d'écriture correspondent à des appels de procédure et que les paramètres effectifs de ces deux procédures sont de type « donnée ».

2.4 AFFECTATION D'UN NOM À UN OBJET

Nous avons vu qu'il était possible de donner un nom à un objet particulier, soit par une action de lecture, soit par l'intermédiaire d'un paramètre « résultat » d'une procédure.

Est-ce que la relation établie entre un nom et l'objet qu'il désigne reste vérifiée tout au long de l'exécution du programme ? Cela dépend du programme. Cette relation peut rester vérifiée durant toute l'exécution du programme, mais aussi cesser. Considérons les deux lectures consécutives suivantes :

```
lire(x)
lire(x)
```

Après la seconde lecture, la première relation entre x et l'objet lu a cessé, et une seconde a été établie entre le même nom x et le second objet lu sur l'entrée standard. Un nom qui sert à désigner un ou plusieurs objets s'appelle une *variable*.

Il est pourtant utile ou nécessaire, en particulier pour des raisons de fiabilité du programme, de garantir qu'un nom désigne toujours le même objet en tout point du programme.

Un nom qui ne peut désigner qu'un seul objet, c'est-à-dire que la relation qui les lie ne peut être remise en cause, s'appelle une *constante*.

Y-a-t-il d'autres façons d'établir cette relation que les deux que nous venons d'indiquer ? La réponse est affirmative. Presque tous les langages de programmation possèdent une action élémentaire, appelée *affectation*, qui associe un nom à un objet.

Chaque langage de programmation a sa manière de concevoir et de représenter l'action d'affectation, mais cette action comporte toujours trois parties : le nom choisi, l'objet à désigner et le signe opératoire identifiant l'action d'affecter. Dans un langage comme PASCAL, le signe d'affectation est : =. L'exemple suivant montre deux actions d'affectation consécutives :

```
x:=6;
y:=x
```

Il faut bien comprendre que `y:=x` signifie « faire désigner par `y` le même objet que celui désigné par `x` », en l'occurrence 6, et non pas « faire que les noms `x` et `y` soient les mêmes ».

Dans notre notation algorithmique, nous choisirons le signe opératoire ← pour représenter l'affectation.

2.5 DÉCLARATION D'UN NOM

Pour des raisons de sécurité des programmes construits, certains langages de programmation exigent que les noms qui servent à désigner les objets soient *déclarés*. C'est le cas par exemple en JAVA où tous les noms (non prédéfinis) doivent avoir été déclarés *au préalable* à l'aide de commandes de déclaration. Toutefois, les noms prédéfinis peuvent être utilisés tels quels sans déclaration préalable.

Afin d'accroître la lisibilité des programmes (même pour des programmes de petite taille), les noms choisis doivent être « significatifs » (et certainement longs), c'est-à-dire qu'ils possèdent un sens qui exprime clairement leur utilisation ultérieure. Si nécessaire, un commentaire peut accompagner la déclaration pour donner plus de précision. Notez toutefois que dans le cas de noms conventionnels, une seule lettre peut suffire. Par exemple, on notera `a`, `b` et `c` les trois coefficients d'une équation du second degré, et souvent `i` l'indice d'une boucle (voir le chapitre 8).

2.5.1 Déclaration de constantes

Une déclaration de constante établit un lien définitif entre un nom et une valeur particulière. Ce nom sera appelé *identificateur de constante*. L'exemple qui suit présente la déclaration de deux constantes :

```
constantes
        nblettres = 26 {nombre de lettres dans l'alphabet latin}
        nbtours   = 33 {nombre de tours par minute}
```

2.5.2 Déclaration de variables

Une déclaration de variable établit un lien entre un nom et un ensemble de valeurs. Le nom ne pourra désigner que des valeurs prises dans cet ensemble. Ce nom s'appelle un *identificateur de variable* et l'ensemble de valeurs un *type*. Cette dernière notion sera présentée dans le chapitre suivant. Dans la déclaration de variables qui suit, le domaine de valeur (introduit par le mot **type**, voir le chapitre 3) nom `réponse` est un ensemble de caractères, alors que celui des noms `racine1` et `racine2` est l'ensemble des réels.

```
variables
     réponse type caractère
     racine1, racine2 type réel
```

2.6 RÈGLES DE DÉDUCTION

L'appel de procédure et l'affectation sont les deux premières actions élémentaires dont nous allons définir les règles de déduction. Pour vérifier la validité de nos programmes, il nous faut donner les règles de déduction de ces deux actions, c'est-à-dire pouvoir déterminer le *conséquent* en fonction de l'*antécédent* par application de l'action d'affectation ou d'appel de procédure.

2.6.1 L'affectation

Pour une affectation `x ← e`, la règle de passage de l'antécédent au conséquent s'exprime de la façon suivante :

$\{ A \}$ `x ← e` $\{ A_e^x \}$

Ce qui peut se traduire par : dans l'antécédent A remplacez toutes les apparitions libres[1] de `e` par `x`. Par exemple, supposons qu'une variable `i` soit égale à 10, que vaut `i` après l'affectation `i ← i+1` ? De toute évidence 11. Montrons-le en faisant apparaître la partie droite de l'affectation dans l'antécédent, puis en appliquant la règle de déduction :

```
{i = 10}
{i + 1 = 10 + 1}
i ← i + 1
{i = 10 + 1 = 11}
```

Réciproquement, on déduit l'antécédent du conséquent en remplaçant dans le conséquent toutes les apparitions de `x` par `e`.

$\{ A_e^x \}$ `x ← e` $\{ A \}$

Dans l'exemple suivant, il faut lire de bas en haut à partir du conséquent.

```
{x + y = 10}
z ← x + y
{z = 10}
```

1. L'expression `e` est sans *effet de bord*, c'est-à-dire qu'elle ne modifie pas son environnement.

Considérons maintenant les quatre affectations suivantes :

```
d ← d + 2
y ← y + d
d ← d + 2
y ← y + d
```

Que calcule cette série d'affectations, lorsque l'antécédent initial est égal à :

$\{y = x^2,\ d = 2x - 1\}$

L'application des règles de déduction montre que la variable y prend successivement les valeurs x^2, $(x+1)^2$ et $(x+2)^2$. Notez qu'en poursuivant, on calcule la suite $(x+i)^2$.

$\{y = x^2,\ d = 2x - 1\}$
d ← d + 2
$\{y + d = (x + 1)^2,\ d = 2x + 1\}$
y ← y + d
$\{y = (x + 1)^2,\ d = 2x + 1\}$
d ← d + 2
$\{y + d = (x + 2)^2),\ d = 2x + 3\}$
y ← y + d
$\{y = (x + 2)^2,\ d = 2x + 3\}$

Tel qu'il est traité, cet exemple applique les règles *a posteriori*. Rappelons, même si cela est difficile, que les affirmations doivent être construites *a priori*, ou du moins simultanément avec le programme.

2.6.2 L'appel de procédure

Les règles de passage de l'antécédent au conséquent (et réciproquement) dépendent des paramètres et du rôle de la procédure. Nous verrons comment décrire ces règles plus précisément dans le chapitre 6.

2.7 LE PROGRAMME SINUS ÉCRIT EN JAVA

Comment s'écrit en JAVA, le programme qui lit un entier sur l'entrée standard, qui calcule et écrit son sinus sur la sortie standard ? Rappelons tout d'abord l'algorithme.

```
Algorithme Sinus
   variable x type entier
   {il existe une donnée à lire sur l'entrée standard}
   lire(x)
   {une donnée a été lue sur l'entrée standard,
    placée en mémoire centrale et le nom x permet de la désigner}
   écrire(sin(x))
   {la valeur du sinus de x est écrite sur la sortie standard}
```

La programmation en JAVA de cet algorithme est la suivante :

```
/** La classe Sinus calcule et affiche sur la sortie standard
    le sinus d'une donnée lue sur l'entrée standard */
import java.io.*;
class Sinus {
   public static void main (String[] args) throws IOException
   {
      int x;
      // il existe une donnée à lire sur l'entrée standard
      x = StdInput.readlnInt();
      // une donnée a été lue sur l'entrée standard,
      // placée en mémoire centrale et
      // le nom x permet de la désigner
      System.out.println(Math.sin(x));
      // la valeur du sinus de x est écrite sur la sortie standard
   }
} // fin classe Sinus
```

Ce premier programme comporte un certain nombre de choses mystérieuses et qui le resteront encore un peu.

Mais, sachez dès à présent, que la structuration d'un programme JAVA est faite autour des objets. Un programme JAVA est une collection de classes (voir plus loin le chapitre 7), placée dans des fichiers de texte, qui décrit des objets manipulés lors de son exécution. Il doit posséder au moins une classe, ici `Sinus`, contenant la procédure `main` par laquelle débutera l'exécution du programme. Ici, cette classe sera conservée dans un fichier qui porte le nom de la classe suffixé par `java`, *i.e.* `Sinus.java`. Par convention, la première lettre de chaque mot qui forment le nom de la classe est en majuscule.

Ce premier programme comporte en tête un commentaire qui explique de façon concise son rôle. C'est une bonne habitude de programmation que de mettre systématiquement une telle information, qui pourra être complétée par le ou les noms des auteurs et la date de création du programme.

Le corps de la procédure `main`, placé entre deux accolades, déclare en premier lieu, la variable entière `x`. Les variables sont déclarées en tête de procédure sans mot-clé particulier pour les introduire. Remarquez que le nom du type précède le nom de la variable. Si le mot-clé **`final`** précède la déclaration, alors il s'agit d'une définition de constante (notez qu'aucune constante n'est déclarée dans ce premier programme). Par exemple, on déclarera la constante réelle `pi` de la façon suivante :

```
final double pi = 3.1415926;
```

La constante prend une valeur lors de sa déclaration et ne pourra évidemment plus être modifiée par la suite.

La lecture est faite grâce à la fonction `readlnInt` [2], qui lit sur l'entrée standard une suite de chiffres, sous forme de caractères, et renvoie la valeur du nombre entier qu'elle représente. Cet entier est ensuite affecté à la variable `x`. Vous noterez que le symbole d'affectation est le

2. Cette fonction n'appartient pas à l'environnement standard de JAVA (voir la section 14.6 page 155).

signe égal (=). Attention de ne pas le confondre avec l'opérateur d'égalité représenté par un autre symbole ! Ce choix du symbole mathématique d'égalité est une aberration héritée du langage C [ANS89].

Le dernier énoncé calcule et écrit le sinus de `x` grâce, respectivement, à la fonction mathématique `sin` et à la procédure `println`.

Les affirmations sont dénotées sous forme de *commentaires* introduits par deux barres obliques `//`. Remarquez la différence de notation avec le premier commentaire en tête de programme. Le langage JAVA propose trois formes de commentaires. Nous distinguerons les commentaires destinés au programmeur de la classe et ceux destinés à l'utilisateur de la classe.

Les premiers débutent par `//` et s'achèvent à la fin de la ligne, ou peuvent être rédigés sur plusieurs lignes entre les parenthéseurs `/*` et `*/`. Ils décrivent en particulier les affirmations qui démontrent la validité du programme.

Les seconds, destinés aux utilisateurs de la classe, sont appelés commentaires de *documentation*. Ils apparaissent entre `/**` et `*/` et peuvent être traités automatiquement par un outil, `javadoc`, pour produire la documentation du programme au format HTML (*Hyper Text Markup Language*).

2.8 EXERCICES

Exercice 2.1. Modifiez le programme pour rendre l'utilisation de la variable `x` inutile.

Exercice 2.2. En partant de l'antécédent *{fact = i!}*, appliquez la règle de déduction de l'énoncé d'affectation pour trouver le conséquent des deux instructions suivantes :

```
i ← i+1
fact ← fact*i
```

Chapitre 3

Types élémentaires

Une façon de distinguer les objets est de les classer en fonction des actions qu'on peut leur appliquer. Les classes obtenues en répertoriant les différentes actions possibles, et en mettant dans la même classe les objets qui peuvent être soumis aux mêmes actions s'appellent des *types*. Classiquement, on distingue deux catégories de type : les types élémentaires et les types structurés. Dans ce chapitre, nous n'étudierons que les objets élémentaires.

On dira qu'un type est *élémentaire*, ou de type simple, si les actions qui le manipulent ne peuvent accéder à l'objet que dans sa totalité.

Le plus souvent, les types élémentaires sont prédéfinis par le langage, c'est-à-dire qu'ils préexistent, et sont directement utilisables par le programmeur. Il s'agit, par exemple, des types entier, réel, booléen ou caractère.

Le programmeur peut également définir ses propres types élémentaires, en particulier pour spécifier un domaine de valeur particulier. Certains langages de programmation offrent pour cela des *constructeurs* de types élémentaires.

Un langage est dit *typé* si les variables sont associées à un type particulier lors de leur déclaration. Pour ces langages, les compilateurs peuvent alors vérifier la cohérence des types des variables, et ainsi garantir une plus grande fiabilité des programmes construits. Au contraire, les variables des langages de programmation *non typés* peuvent désigner des objets de n'importe quel type et les vérifications de cohérence de type sont reportées au moment de l'exécution du programme. La programmation avec ces langages est moins sûre, mais offre plus de souplesse.

Dans ce chapitre, nous présenterons les types élémentaires entier, réel, booléen et caractère, ainsi que les constructeurs de type énuméré et intervalle.

3.1 LE TYPE ENTIER

Le type *entier* représente partiellement l'ensemble des entiers relatifs $\mathbb{Z}$ des mathématiciens. Alors que l'ensemble $\mathbb{Z}$ est infini, l'ensemble des valeurs défini par le type entier est *fini*, et limité par les possibilités de chaque ordinateur, en fait, par le nombre de bits utilisés pour sa représentation. Le type entier possède donc un élément minimum et un élément maximum. Chaque entier possède une représentation distincte sur l'ordinateur et la notation des constantes entières est en général classique : une suite de chiffres en base 10.

La cardinalité du type entier dépend de la représentation binaire des nombres. Un mot de n bits permet de représenter 2^n nombres positifs sur l'intervalle $[0,2^n - 1]$. Réciproquement, le nombre de bits nécessaires à la représentation d'un entier n est $\log_2 n$. Afin de simplifier les opérations d'addition et de soustraction, les entiers négatifs sont représentés sous forme complémentée, soit en *complément à un* , soit en *complément à deux*. En complément à un, la valeur négative d'un entier x est obtenue en inversant chaque position binaire de sa représentation. Par exemple, sur 4 bits l'entier 6 est représenté par 0110 et l'entier -6 par la configuration binaire 1001. On obtient le complément à deux, en ajoutant 1 au complément à un. L'entier -6 est donc représenté par 1010. En complément à un, l'ensemble des entiers est défini par l'intervalle $[-2^{n-1} - 1,2^{n-1} - 1]$, où n est le nombre de bits utilisés pour représenter un entier. Notez qu'en complément à un, l'entier zéro possède deux représentations binaires, la première avec tous les bits à 0 et la seconde avec tous les bits à 1. En complément à deux, le type entier est défini par l'intervalle $[-2^n - 1,2^{n-1} - 1]$ et il n'existe qu'une seule représentation du zéro. Une configuration binaire dont tous les bits valent un représente l'entier -1.

Les opérations de l'arithmétique classique s'appliquent sur le type entier, de même que les opérations de comparaison. Notez que les axiomes ordinaires de l'arithmétique entière ne sont pas valables sur l'ordinateur car ils ne sont pas vérifiés quand on sort du domaine de définition des entiers. En particulier, l'addition n'est pas une loi associative sur le type entier. Si $entmax$ est l'entier maximum et x un entier positif, la somme $(entmax + 1) - x$ n'est pas définie, alors que $entmax + (1 - x)$ appartient au domaine de définition.

➤ Les types entiers de JAVA

Tout d'abord, notons qu'il n'existe pas un, mais quatre types entiers. Les types entiers **`byte`**, **`short`**, **`int`**, **`long`** sont signés et représentés en complément à 2. Ils se distinguent par leur cardinal. Plus précisément, les entiers du type **`byte`** sont représentés sur 8 bits, ceux du type **`short`** sur 16 bits, ceux du type **`int`** sur 32 bits et ceux du type **`long`** sur 64 bits.

Pour chacun de ces quatre types, il existe deux constantes qui représentent l'entier minimum et l'entier maximum. Ces constantes sont données par la table 3.1.

Les constantes entières en base 10 sont dénotées par une suite de chiffres compris entre `0` et `9`. Le langage permet également d'exprimer des valeurs en base 8 ou 16, en les faisant précéder, respectivement, par les préfixes `0` et `0x`.

```
{exemples de constantes entières}
3 125 0 0777 3456 234 0xAC12
```

type	*minimum*	*maximum*
byte	`Byte.MIN_VALUE`	`Byte.MAX_VALUE`
short	`Short.MIN_VALUE`	`Short.MAX_VALUE`
int	`Integer.MIN_VALUE`	`Integer.MAX_VALUE`
long	`Long.MIN_VALUE`	`Long.MAX_VALUE`

TAB. 3.1 – *Valeurs minimales et maximales des types entiers de* JAVA .

La table 3.2 montre les opérateurs arithmétiques et relationnels qui peuvent être appliqués sur les types entiers de JAVA.

	opérateur	*fonction*	*exemple*
opérateurs arithmétiques			
	`-`	opposé	`-45`
	`+`	addition	`45 + 5`
	`-`	soustraction	`a - 4`
	`*`	multiplication	`a * b`
	`/`	division	`5 / 45`
	`%`	modulo	`a % 4`
opérateurs relationnels			
	`<`	inférieur	`a < b`
	`<=`	inférieur ou égal	`a <= b`
	`==`	égal	`a == b`
	`!=`	différent	`a != b`
	`>`	supérieur	`a > b`
	`>=`	supérieur ou égal	`a >= b`

TAB. 3.2 – *Opérateurs sur les types entiers.*

La déclaration d'une variable entière est précédée du domaine de valeur particulier qu'elle peut prendre. Notez que plusieurs variables d'un même type peuvent être déclarées en même temps.

```
byte unOctect, uneNote;
short nbMots;
int population;
long nbÉtoiles, infini;
```

3.2 LE TYPE RÉEL

Le type *réel* sert à définir partiellement l'ensemble $\mathbb{R}$ des mathématiciens. Les réels ne peuvent être représentés sur l'ordinateur que par des *approximations* plus ou moins fidèles. Le type réel décrit un nombre fini de représentants d'intervalles du *continuum* des réels. Si deux objets réels sont dans le même intervalle, ils auront le même représentant et ne pourront pas être distingués. De plus, les réels ne sont pas uniformément répartis sur l'ensemble ; plus de la moitié est concentrée sur l'intervalle $[-1,1]$.

Notez que le type entier n'est pas inclus dans le type réel. Ils forment deux ensembles disjoints. La dénotation d'une constante réelle varie d'un langage de programmation à l'autre et nous verrons plus loin le cas particulier de JAVA.

L'arithmétique sur les réels est *inexacte*. Chaque opération conduit à des résultats approchés qui, répétée plusieurs fois, peut conduire à des résultats totalement faux. Les résultats dépendent en effet de la représentation du nombre réel sur l'ordinateur, de la précision avec laquelle il est obtenu ainsi que de la méthode utilisée pour les calculer.

Il existe plusieurs modes de représentation des réels. La plus courante est celle dite en *virgule flottante*. Un nombre réel x est représenté par un triplet d'entiers (s,e,m), tel que $x = (-1)^s \times m \times B^e$, avec $s \in \{0,1\}$, $-E < e < E$ et $-M < m < M$. La valeur de s donne le signe du réel, m s'appelle la *mantisse*, e l'*exposant* et B la *base* (le plus souvent 2, 8 ou 16). E, M et B constituent les caractéristiques de la représentation choisie, et dépendent de l'ordinateur. L'imprécision de la représentation provient des valeurs limites E et M. De plus, les langages de programmation raisonnent en termes de nombres en base 10, alors que ces nombres sont représentés dans des bases différentes.

Les opérateurs d'arithmétique réelle et de relation sont applicables sur les réels, mais il faut tenir compte du fait qu'elles peuvent conduire à des résultats faux. En particulier, le test de l'égalité de deux nombres réels est à bannir. Comme pour le type entier, ces opérations ne sont pas des lois de composition internes.

La représentation d'un nombre en virgule flottante n'est pas unique tant que l'emplacement du délimiteur dans la mantisse n'est pas défini, puisqu'un décalage du délimiteur peut être compensé par une modification de l'exposant. Ainsi, 125.32 peut s'écrire $0.12532 * 10^3$, $1.2532 * 10^2$ ou encore $12.532 * 10^1$. Pour lever cette ambiguïté, on adopte généralement une représentation normalisée [1]. Les réels sont tels que la valeur de l'exposant est ajustée pour que la mantisse ait le plus de chiffres significatifs possibles. Après chaque opération, le résultat obtenu est normalisé.

Les exemples suivants mettent en évidence l'inexactitude des calculs réels. Pour simplifier, on considère que les nombres réels sont représentés en base 10 et que la mantisse ne peut utiliser que quatre chiffres ($M = 1000$). Soient les réels x, y et z suivants :

$$x = 9.900, y = 1.000, z = -0.999$$

on veut calculer $(x+y)+z$ et $x+(y+z)$. Les erreurs de calculs viennent des ajustements de représentation. Par exemple, lors d'une addition de deux entiers, on ajuste la représentation du nombre qui a le plus petit exposant en valeur absolue de façon que les deux exposants soient égaux. On note $\bar{x}$ la représentation informatique de x.

$$\bar{x} = 9900\ 10^{-3}, \bar{y} = 1000\ 10^{-3}, \bar{z} = -9990\ 10^{-4}$$

$$\bar{x} + \bar{y} = 10900\ 10^{-3} = 1090\ 10^{-2}$$
$$(\bar{x} + \bar{y}) + \bar{z} = 1090\ 10^{-2} + -99\ 10^{-2} = 991\ 10^{-2} = 9910\ 10^{-3}$$

$$\bar{y} + \bar{z} = 1000\ 10^{-3} + -9990\ 10^{-4} = 1000\ 10^{-3} + -999\ 10^{-3} = 1\ 10^{-3}$$
$$\bar{x} + (\bar{y} + \bar{z}) = 9900\ 10^{-3} + 1\ 10^{-3} = 9901\ 10^{-3}$$

1. La norme IEEE 754 propose une représentation normalisée des réels sur 32, 64, et 128 bits. Cette norme est aujourd'hui très utilisée.

Seul le calcul $\bar{x} + (\bar{y} + \bar{z})$ est juste. L'erreur, dans le calcul de $(\bar{x} + \bar{y}) + \bar{z}$, provient de la perte d'un chiffre significatif de $\bar{z}$ dans l'ajustement à -2.

Considérons maintenant les trois réels $x = 1100$, $y = -5$, $z = 5.001$. On désire calculer $\bar{x} \times (\bar{y} + \bar{z})$ et $(\bar{x} \times \bar{y}) + (\bar{x} \times \bar{z})$.

$$\bar{x} = 1100\ 10^0, \bar{y} = -5000\ 10^{-3}, \bar{z} = 5001\ 10^{-3}$$

$$\bar{x} \times \bar{y} = -5500\ 10^0$$
$$\bar{x} \times \bar{z} = 5501\ 10^0$$
$$(\bar{x} \times \bar{y}) + (\bar{x} \times \bar{z}) = 1000\ 10^{-3} = 1$$

$$\bar{y} + \bar{z} = 1000\ 10^{-6}$$
$$\bar{x} \times (\bar{y} + \bar{z}) = 1100\ 10^{-3} = 1.1$$

Là encore, seul le deuxième calcul est juste. Enfin, traitons la résolution d'une équation du second degré avec les trois coefficients réels $a = 1$, $b = -200$, $c = 1$. Nous obtenons :

$$\bar{a} = 1000\ 10^{-3}, \bar{b} = -2000\ 10^{-1}, \bar{c} = 1000\ 10^{-3}$$

$$\Delta = \sqrt{4000\ 10^1 - 4000\ 10^{-3}} = 2000\ 10^{-1}$$

Dans le calcul de Δ, la disparition de $4000\ 10^{-3}$ au cours de l'ajustement conduit au calcul d'une racine fausse. On trouve $\bar{x}_1 = 0$ et $\bar{x}_2 = 2000\ 10{-}1$, alors que la racine $\bar{x}_1$ est normalement égale à 0.005.

Dans la réalité, la mantisse est beaucoup plus grande et les calculs plus précis, mais les problèmes restent identiques. D'une façon générale, les opérations d'addition et de soustraction sont dangereuses lorsque les opérandes ont des valeurs voisines qui conduisent à un résultat proche de zéro, de même la division, lorsque le dénominateur est proche de zéro.

Les types réels de JAVA

Comme pour les entiers, JAVA définit plusieurs types réels : **`float`** et **`double`**. Les réels du type **`float`** sont en simple précision codés sur 32 bits, ceux du type **`double`** sont en double précision codés sur 64 bits. La représentation de ces réels suit la norme IEEE 754.

Ces deux types possèdent chacun une valeur minimale et une valeur maximale données par le tableau 3.3.

type	*minimum*	*maximum*
`float`	`Float.MIN_VALUE`	`Float.MAX_VALUE`
`double`	`Double.MIN_VALUE`	`Double.MAX_VALUE`

TAB. 3.3 – *Valeurs minimales et maximales des types réels de* JAVA .

La dénotation d'une constante réelle suit la syntaxe donnée ci-dessous. Les crochets indiquent que l'argument qu'ils entourent est optionnel, la barre verticale un choix, et `entier` est un nombre entier en base 10.

```
entier[.][entier][[e|E][±]entier]
[entier][.]entier[[e|E][±]entier]
```

Ainsi, les constantes suivantes sont valides :

```
123.456   34.0   .0   12.   56e34  4E-5 45.67e2  4567
```

La table 3.4 donne les opérateurs arithmétiques et relationnels applicables sur les types réels de JAVA.

	opérateur	*fonction*	*exemple*
opérateurs arithmétiques			
	`-`	opposé	`-45.5+5e12`
	`+`	addition	`45.5+5e12`
	`-`	soustraction	`a - 4.3`
	`*`	multiplication	`a * b`
	`/`	division	`5 / 45`
opérateurs relationnels			
	`<`	inférieur	`a < b`
	`<=`	inférieur ou égal	`a <= b`
	`==`	égal	`a == b`
	`!=`	différent	`a != b`
	`>`	supérieur	`a > b`
	`>=`	supérieur ou égal	`a >= b`

TAB. 3.4 – *Opérateurs sur les types réels.*

Enfin, les déclarations de variables de type réel ont la forme suivante :

```
float distance, rayon;
double surface;
```

3.3 LE TYPE BOOLÉEN

Le type *booléen* est un type fini qui représente un ensemble composé de deux valeurs logiques, *vrai* et *faux*, sur lequel les opérations de disjonction (*ou*), de disjonction exclusive (*xou*), de conjonction (*et*), et de négation (*non*) peuvent être appliquées. Ces opérations, que l'on trouve dans la plupart des langages de programmation, sont entièrement définies au moyen de la table 3.5, dite de *vérité*.

p	q	$non\ p$	$p\ et\ q$	$p\ ou\ q$	$p\ xou\ q$
faux	faux	vrai	faux	faux	faux
vrai	faux	faux	faux	vrai	vrai
faux	vrai	vrai	faux	vrai	vrai
vrai	vrai	faux	vrai	vrai	faux

TAB. 3.5 – *Table de vérité.*

À partir de cette table, on peut déduire un certain nombre de relations, et en particulier celles de DE MORGAN qu'il est fréquent d'appliquer :

$$non\ (p\ ou\ q) = non\ p\ et\ non\ q$$
$$non\ (p\ et\ q) = non\ p\ ou\ non\ q$$

Notez qu'un seul bit est nécessaire pour la représentation d'une valeur booléenne. Mais, le plus souvent, un booléen est codé sur un octet avec, par convention, 0 pour la valeur *faux* et 1 pour la valeur *vrai*.

Le type booléen de JAVA

Le type **`boolean`** définit les deux valeurs **`true`** et **`false`**. En plus des opérateurs de négation (`!`), de disjonction (`|`), de disjonction exclusive (`^`) et de conjonction (`&`), JAVA propose les opérateurs de disjonction et de conjonction conditionnelle représentés par les symboles `||` et `&&`. Le résultat de `p || q` est vrai si `p` est vrai quelle que soit la valeur de `q` qui, dans ce cas, n'est pas évaluée. De même, le résultat de `p && q` est faux si `p` est faux quelle que soit la valeur de `q` qui, dans ce cas, n'est pas évaluée. Nous verrons ultérieurement que l'utilisation de ces opérateurs a une influence considérable sur le style de programmation.

La déclaration suivante est un exemple de déclaration de variables booléennes.

```
boolean présent, onContinue;
```

3.4 LE TYPE CARACTÈRE

Chaque ordinateur possède son propre jeu de caractères. La plupart des ordinateurs actuels proposent plusieurs jeux de caractères différents et normalisés pour représenter des lettres et des chiffres de diverses langues, des symboles graphiques ou des caractères de contrôle.

Par le passé, seuls deux jeux de caractères américains étaient vraiment disponibles. Le jeu de caractères ASCII[2], codé sur 7 bits, ne permet de représenter que 128 caractères différents. Le jeu EBCDIC[3], spécifique à IBM, code les caractères sur 8 bits et inclut quelques lettres étrangères, comme par exemple, ß ou ü.

2. ASCII est l'acronyme de *American Standard Code for Information Interchange*. Ce jeu de caractères est une norme ISO (*International Organization for Standardization*).

3. EBCDIC est l'acronyme de *Extended Binary Coded Decimal*.

Pour satisfaire les usagers non anglophones, la norme ISO-8859 propose plusieurs jeux de 256 caractères codés sur 8 bits. Les 128 premiers caractères sont ceux du jeux ASCII et les 128 suivants correspondent à des variantes nationales. La norme ISO 8859-1, appelée Latin-1, correspond à la variante des pays de l'Europe de l'Ouest. Elle inclut des symboles graphiques ainsi que des caractères à signes diacritiques comme é, à, ç ou encore å, qui existent à la fois sous forme minuscule et majuscule (sauf ß et ÿ).

Mais, face à l'internationalisation toujours croissante de l'informatique, ces jeux de caractères ne suffisent plus pour représenter tous les symboles des différentes langues du monde. L'ISO a proposé une norme pour représenter des caractères sur 32 bits, permettant ainsi de définir plus de 4 milliards de caractères différents. Ce nombre très important pose naturellement des problèmes techniques aux créateurs de polices de caractères, et seul un sous-ensemble de ce jeu codé sur 16 bits, appelé UNICODE, commence à être exploité par quelques systèmes d'exploitation et langages de programmation. Les 256 premiers caractères du jeu UNICODE sont ceux de la norme ISO 8859, les suivants représentent entre autres des symboles de langages variés (dont le braille), des symboles mathématiques ou encore des symboles graphiques (*e.g.* dingbats). La description complète du jeu UNICODE est accessible à l'adresse `charts.unicode.org`.

Dans tous les langages de programmation, il existe une relation d'ordre sur les caractères, qui fait apparaître une bijection entre le type caractère et l'intervalle d'entiers [0,*ordmaxcar*], où *ordmaxcar* est l'ordinal (*i.e.* le numéro d'ordre) du dernier caractère du jeu de caractères. On peut donc appliquer les opérateurs relationnels sur les objets de type caractère.

Le type caractère de JAVA

Le type **`char`** utilise le jeu de caractères UNICODE. Les constantes de type caractère sont dénotées entre deux apostrophes. Ainsi, les caractères `'a'` et `'ç'` représentent les lettres alphabétiques a et ç. Il est fondamental de comprendre la différence entre les notations `'2'` et `2`. La première représente un caractère et la seconde un entier.

Certains caractères non imprimables possèdent une représentation particulière. Une partie de ces caractères est donnée dans la table 3.6.

Caractère	*Notation*
passage à la ligne	`\n`
tabulation	`\t`
retour en arrière	`\r`
saut de page	`\f`
backslash	`\\`
apostrophe	`\'`

TAB. 3.6 – *Caractères spéciaux.*

Il existe une relation d'ordre sur le type **`char`**. On peut donc appliquer les opérateurs de relation <, <=, =, !=, > et >= sur des opérandes de type caractères.

Les caractères peuvent être dénotés par la valeur hexadécimale de leur ordinal, précédée par la lettre `u` et le symbole `\`. Par exemple, `'\u0041'` est le caractère d'ordinal 65, c'est-à-dire la lettre A. Cette notation particulière trouve tout son intérêt lorsqu'il s'agit de dénoter des

caractères graphiques. Par exemple, les caractères '\u12cc' et '\u1356' représentent les lettres éthiopiennes ጜ et ፖ, les caractères '\u2200' et '\u2208' représentent les symboles mathématiques ∀ et ∈, et '\u2708' est le symbole ✈. La fonction `Character.getNumericValue` retourne l'ordinal du caractère UNICODE passé en paramètre.

La déclaration suivante introduit trois nouvelles variables de type caractère :

```
char lettre, marque, symbole;
```

Notez que les caractères UNICODE peuvent être normalement utilisés dans la rédaction des programmes JAVA pour dénoter des noms de variables ou de fonctions. Afin d'accroître la lisibilité des programmes, il est fortement conseillé d'utiliser les caractères accentués, s'ils sont nécessaires. Il est aussi possible d'utiliser toutes sortes de symboles, et la déclaration de constante suivante est tout à fait valide :

```
final double π = 3.1415926;
```

Si la saisie directe du symbole π n'est pas possible, il sera toujours possible d'employer la notation hexadécimale.

```
final double \u03C0 = 3.1415926;
```

3.5 CONSTRUCTEURS DE TYPES SIMPLES

Les types *énumérés* et *intervalles* permettent de construire des ensembles de valeurs particulières. L'intérêt de ces types est de pouvoir spécifier précisément le domaine de définition des variables utilisées dans le programme. Certains langages de programmation, mais pas JAVA, proposent de tels constructeurs et permettent de nommer les types élémentaires construits. Dans cette section, nous présentons succinctement une notation algorithmique pour définir des types énumérés et intervalles qui nous serviront par la suite.

➢ Les types énumérés

Une façon simple de construire un type est d'énumérer les éléments qui le composent. On indique le nom du type suivi, entre accolades, des valeurs de l'ensemble à construire. Ces valeurs sont des noms de constantes ou des constantes prises dans un même type élémentaire. L'exemple suivant montre la déclaration de trois types énumérés.

```
couleurs = ( vert, bleu, gris, rouge, jaune )
nbpremiers = ( 1, 3, 5, 7, 11, 13 )
voyelles = ( 'a', 'e', 'i', 'o', 'u', 'y' )
```

Il existe une relation d'ordre sur les types énumérés et, d'une façon générale, tous les opérateurs relationnels sont applicables sur les types énumérés.

➢ Les types intervalles

Un type intervalle définit un intervalle de valeurs sur un type de base. Les types de base possibles sont les types élémentaires et la déclaration doit indiquer les bornes inférieure et supérieure de l'intervalle. La forme générale d'une déclaration d'un intervalle est la suivante :

```
[binf,bsup]
```

Les bornes inférieure et supérieure sont des constantes de même type. Les opérations possibles sur les intervalles sont celles admises sur le type de base. Les déclarations suivantes définissent un type entier naturel et un type lettre alphabétique. L'exemple suivant montre la déclaration de deux types intervalles.

```
naturel = [0,entmax]
lettres = ['a','z']
```

3.6 EXERCICES

Exercice 3.1. Parmi les notations de constantes JAVA suivantes, indiquez celles qui sont valides, ainsi que le type des nombres :

0.31	+273.3	0.005e+3	0x10
010	.389	15	0x5e-4
33.75	1.5+2	3,250	.E1
1234	3E5	08	10e-4
0X1a2	0037	1e2768	0x1A2

Exercice 3.2. Indiquez le type de chacune des constantes JAVA données ci-dessous :

100	**true**	'a'	2
0x10	.23	"nom"	'2'
"a"	'\u0041'	'\n'	"2"

Exercice 3.3. Est-ce que la disjonction et la conjonction sont des lois commutatives et associatives ? En d'autres termes, les égalités suivantes sont-elles vérifiées ?

$$(p \; ou \; q) \; ou \; r = p \; ou \; (q \; ou \; r)$$
$$(p \; et \; q) \; et \; r = p \; et \; (q \; et \; r)$$

Exercice 3.4. Est-ce que la disjonction est distributive par rapport à la conjonction ? Réciproquement, la conjonction est-elle distributive par rapport à la disjonction ? Vérifiez les égalités suivantes :

$$(p \; ou \; q) \; et \; r = (p \; et \; q) \; ou \; (q \; et \; r)$$
$$(p \; et \; q) \; ou \; r = (p \; ou \; q) \; et \; (q \; ou \; r)$$

Exercice 3.5. On dit qu'il y a dépassement de capacité, lorsque les opérations arithmétiques produisent des résultats en dehors de leur ensemble de définition. Certains langages de programmation signalent les dépassements de capacité d'autres pas. Vérifiez expérimentalement l'attitude de JAVA en cas de dépassement de capacité pour des opérandes de type entier et réel.

Chapitre 4

Expressions

Comme le langage mathématique, les langages de programmation permettent de composer entre eux des *opérandes* et des *opérateurs* pour former des *expressions*. Les opérandes sont des valeurs ou des noms qui donnent accès à une valeur. Ces sont bien évidemment des identificateurs de constantes ou de variables, mais aussi des appels de fonctions. Les opérateurs correspondent à des opérations qui portent sur un ou plusieurs opérandes. Les opérateurs *unaires* ou *monadiques* possèdent un unique opérande ; les opérateurs *binaires* ou *dyadiques* ont deux opérandes ; ceux qui en possèdent trois sont appelés *ternaires* ou *triadiques*. Un opérateur à n opérandes est dit *n-aire*. Les opérateurs des langages de programmation ont très rarement plus de trois opérandes, et pour un opérateur donné le nombre d'opérandes ne varie jamais.

Dans la plupart des langages, la notation utilisée suit la notation algébrique classique. Cette notation est dite *infixée*, c'est-à-dire que les opérandes se situent de part et d'autre de l'opérateur, comme dans $x + y$ ou encore $x \times y + z$. Elle nécessite des parenthèses pour exprimer par exemple des règles de priorité, $x + y \times z$ est différent de $(x + y) \times z$.

Certains langages, comme LISP, utilisent la notation *polonaise*[1], également appelée *préfixée*, qui place l'opérateur systématiquement avant ses opérandes. On écrit par exemple $+\ x\ y$ ou $\times\ +\ x\ y\ z$. Les parenthèses sont inutiles, $+\ x\ \times\ y\ z$ est différent de $\times\ +\ x\ y\ z$. Remarquez que l'appel d'une fonction ou d'une procédure est à considérer comme une notation préfixée, où l'opérateur est le nom de la fonction ou de la procédure, et ses opérandes sont les paramètres effectifs.

La notation *polonaise inverse*, appelée aussi notation *postfixée*, place l'opérateur à la suite de ses opérandes. Les possesseurs de calculettes HP connaissent bien cette notation qui im-

1. Ainsi appelée car son invention est due au mathématicien polonais JAN ŁUKASIEWICW.

pose une écriture des expressions de la forme $x\ y\ +$ ou $x\ y \times z\ +$. Avec cette notation, les parenthèses sont également inutiles puisque $(x + y)/z$ s'écrit $x\ y\ +\ z\ /$.

Dans la suite de ce chapitre, nous n'utiliserons que la notation infixée dans la mesure où elle est la plus employée par les langages de programmation.

4.1 ÉVALUATION

Le but d'une expression est de calculer, lors de son *évaluation*, un résultat. En général, l'évaluation d'une expression produit un résultat unique. Mais pour certains langages, comme par exemple ICON [GG96], l'évaluation d'une expression peut donner aucun, un ou plusieurs résultats.

Le résultat d'une expression est déterminé par l'ordre d'évaluation des formules simples qui la composent. Cet ordre d'évaluation, pour une même forme syntaxique, n'est pas forcément le même dans tous les langages. En supposant que les opérandes sont tous de même type, nous considérerons trois cas : la composition d'un même opérateur, la composition d'opérateurs différents et l'utilisation de parenthéseurs.

4.1.1 Composition du même opérateur plusieurs fois

L'ordre d'évaluation n'est important que dans la mesure où l'opérateur n'est pas associatif. Dans la plupart des langages de programmation, la grammaire précise que l'ordre d'évaluation est de gauche à droite. C'est le cas pour la majorité des opérateurs ; on dit qu'ils sont *associatifs à gauche*. Ainsi, l'expression $x + y + z$ calcule la somme de $x + y$ et de z.

Plus rarement, les langages proposent des opérateurs *associatifs à droite*. Ceux qui autorisent la composition d'affectations définissent un ordre d'évaluation de droite à gauche de cet opérateur ; $x \leftarrow y \leftarrow z$ commence par affecter à y la valeur de z, puis affecte à x la valeur de y.

4.1.2 Composition de plusieurs opérateurs différents

L'ordre de gauche à droite n'est pas toujours le plus naturel lorsque les opérateurs concernés sont différents. Afin de respecter les habitudes de notation algébrique, la plupart des langages de programmation définissent entre les opérateurs des règles de *priorité* susceptibles de remettre en cause l'ordre d'évaluation de gauche à droite. Par exemple, $x + y \times z$ correspond à l'addition de x et du produit $y \times z$, et non au produit de $x + y$ avec z.

➢ Règles de priorité en JAVA

Les règles de priorité varient considérablement d'un langage de programmation à l'autre, et il est bien difficile de dégager des règles communes. La table 4.1 donne les règles de priorité du langage JAVA, en allant du moins prioritaire au plus prioritaire, des opérateurs vus au chapitre précédent. À niveau de priorité égal, l'évaluation se fait de gauche à droite, sauf pour l'affectation.

opérateur	*fonction*
=	affectation
||	disjonction conditionnelle
&&	conjonction conditionnelle
|	disjonction
&	conjonction
== !=	égalité, inégalité
< > >= <=	opérateurs relationnels
+ -	opérateurs additifs
* / %	opérateurs multiplicatifs
!	négation
( )	sous-expression

TAB. 4.1 – *Règles de priorité des opérateurs* JAVA .

4.1.3 Parenthésage des parties d'une expression

Il arrive que les deux modes de composition précédents ne permettent pas d'exprimer l'ordre dans lequel doit s'effectuer les opérations. Ainsi, si l'on veut additionner x et y, puis multiplier le résultat par z, il est nécessaire de recourir à la notion de *parenthésage*. L'expression s'écrit alors, avec des parenthèses, $(x + y) \times z$.

Les parenthèses permettent de ramener toute sous-expression d'une même expression au rang *d'un et d'un seul* opérande, que l'on peut alors composer comme d'habitude avec le reste de l'expression. Ainsi, dans l'expression $(x + y) \times z$, l'opérateur multiplicatif possède deux opérandes $x + y$ et z.

4.2 TYPE D'UNE EXPRESSION

On vient de voir qu'une expression calcule un résultat. Ce résultat est typé et définit, par voie de conséquence, le type de l'expression. Ainsi, si p et q sont deux booléens, l'expression *p ou q* est une expression booléenne puisqu'elle produit un résultat booléen.

Notez bien qu'une expression peut très bien être formée à partir d'opérateurs manipulant des objets de types différents, et dans ce cas le parenthésage peut servir à délimiter sans ambiguïté les opérandes de même type dont la composition forme un résultat d'un autre type, lui-même opérande d'un autre opérateur dans la même expression. Par exemple, l'expression booléenne suivante :

```
(i ⩽ maxÉléments) et (courant ≠ 0)
```

est formée par les deux opérandes booléens de l'opérateur de conjonction eux-mêmes composés à partir de deux opérandes numériques reliés par des opérateurs de relation à résultat booléen.

4.3 CONVERSIONS DE TYPE

Les objets, mais pas tous, peuvent être convertis d'un type vers un autre. Généralement, on distingue deux types de conversion. Les conversions *implicites* pour lesquelles l'opérateur décide de la conversion à faire en fonction de la nature de l'opérande ; les conversions *explicites*, pour lesquelles le programmeur est responsable de la mise en œuvre de la conversion à l'aide d'une notation adéquate.

Dans les langages fortement typés, les conversions implicites sont, pour des raisons de sécurité de programmation, peu nombreuses. Dans un langage comme PASCAL, il n'existe qu'une seule conversion implicite des entiers vers les réels. En revanche, de par leur nature, les langages non typés ont en général un nombre de conversions implicites importants et il n'est pas toujours simple pour le programmeur de déterminer rapidement le type du résultat de l'évaluation d'une expression.

➢ Les conversions de type en JAVA

Les conversions de type *implicites* sont relativement nombreuses, en partie dues à l'existence de plusieurs types entiers et réels. Ces conversions, appelées également *promotions*, transforment implicitement un objet de type T en un objet d'un type T', lorsque le contexte l'exige. Par exemple, l'évaluation de l'expression $1 + 4.76$ provoquera la conversion implicite de l'entier 1 en un réel double précision 1.0, suivie d'une addition réelle. Les promotions possibles sont données par la table 4.2.

type	*promotion*
`float`	`double`
`long`	`float ou double`
`int`	`long, float ou double`
`short`	`int, long, float ou double`
`char`	`int, long, float ou double`

TAB. 4.2 – *Promotions de type du langage* JAVA .

Les conversions *explicites*, appelées *cast* en anglais, sont faites à l'aide d'un opérateur de conversion. Sa dénotation consiste à placer entre parenthèses, devant la valeur à convertir, le type dans lequel elle doit être convertie. Dans l'exemple suivant, le réel 1.4 est converti explicitement en un entier. Le résultat de la conversion `(int) 1.4` est l'entier 1.

4.4 UN EXEMPLE

On désire écrire un programme qui lit sur l'entrée standard une valeur représentant une somme d'argent et qui calcule et affiche le nombre de billets de 500, 200, 100, 50 et 10 euros qu'elle représente.

L'algorithme

L'algorithme commence par lire sur l'entrée standard l'entier qui représente la somme d'argent et affecte la valeur à une variable `capital`. Pour obtenir la décomposition en nombre de billets et de pièces de la somme d'argent, on procède par des divisions entières successives en conservant chaque fois le reste. La preuve de la validité de l'algorithme s'appuie sur la définition de la division euclidienne :

$\forall a,b \in \mathbb{N}, b > 0$
$a = q \times b + r \text{ et } 0 \leqslant r < b$

Le quotient est obtenu par l'opérateur de division entière, et le reste par celui de modulo. Cet algorithme, avec les assertions qui démontrent la validité du programme, s'exprime comme suit :

Algorithme Somme d'argent
variables capital, reste, b500, b200, b100, b50, b10 **type** entier
{il existe sur l'entrée standard un entier
qui représente une somme d'argent en euros}
lire(capital)
{capital = somme d'argent > 0 et multiple de 10}
b500 $\leftarrow \frac{\text{capital}}{500}$
reste $\leftarrow$ capital modulo 500
{capital = b500×500 + reste}
b200 $\leftarrow \frac{\text{reste}}{200}$
reste $\leftarrow$ reste modulo 200
{capital = b500×500 + b200×200 + reste}
b100 $\leftarrow \frac{\text{reste}}{100}$
reste $\leftarrow$ reste modulo 100
{capital = b500×500 + b200×200 + b100×100 + reste}
b50 $\leftarrow \frac{\text{reste}}{50}$
reste $\leftarrow$ reste modulo 50
{capital = b500×500 + b200×200 + b100×100 + b50×50 + reste}
b10 $\leftarrow \frac{\text{reste}}{10}$
{capital = b500×500 + b200×200 + b100×100 + b50×50 + b10×10}
écrire(b500, b200, b100, b50, b10)
{le nombre de billets de 500, 200, 100, 50 et 10 euros
représentent la valeur de capital}

Le programme en JAVA

À partir de l'algorithme précédent, et des connaissances JAVA déjà acquises, la rédaction du programme ne pose pas de réelles difficultés.

```
/** La classe SommedArgent lit sur l'entrée standard une valeur
 *  représentant une somme d'argent, puis calcule et affiche le
 *  nombre de billets de 500, 200, 100, 50 et 10 euros
 *  qu'elle représente
 */
import java.io.*;
class SommedArgent {
  public static void main (String[] args) throws IOException
  {
      int capital = StdInput.readlnInt(),
          reste, b500, b200, b100, b50, b10;
      // capital = somme d'argent >0 et multiple de 10
      b500 = capital / 500;
      reste = capital % 500;
      // capital = b500×500+reste
      b200 = reste / 200;
      reste %= 200;
      // capital = b500×500+b200×200+reste
      b100 = reste / 100;
      reste %= 100;
      // capital = b500×500+b200×200+b100×100+reste
      b50 = reste / 50;
      reste %= 50;
      // capital = b500×500+b200×200+b100×100+b50×50+reste
      b10 = reste / 10;
      // capital = b500×500+b200×200+b100×100+b50×50+b10×10
      System.out.println(b500+"␣"+b200+"␣"+b100+"␣"+b50+"␣"+b10);
  }
} // fin classe SommedArgent
```

Toutefois, dans ce programme, vous pouvez remarquez deux choses nouvelles. D'une part, l'initialisation de la variable `capital` au moment de sa déclaration, et d'autre part, l'utilisation d'un nouvel opérateur d'affectation `%=`.

Regrouper la déclaration et l'initialisation d'une variable a pour intérêt de clairement localiser ces deux actions. Nous considérerons cela comme une bonne technique de programmation.

L'affectation composée `reste%=200` est équivalente à l'affectation `reste=reste%200`. D'une façon générale, une affectation de la forme `a`*op*`=b` est équivalente à `a=a`*op* `b`, où *op* peut être choisi parmi `+`, `-`, `*`, `/`, `%`, `&`, `|` et encore quelques autres opérateurs dont nous ne parlerons pas pour l'instant. L'intérêt principal de ces opérateurs est de n'évaluer qu'une seule fois l'opérande gauche.

Les affectations de JAVA ont la particularité[2] d'être des expressions, c'est-à-dire de fournir un résultat, en l'occurrence la valeur de l'opérande gauche après affectation. Nos deux premiers programmes JAVA n'utilisent pas cette caractéristique, mais nous verrons un peu plus tard qu'elle influence la façon de programmer les algorithmes.

2. On la trouve également dans les langages C et C++, et plus généralement dans les langages d'expression. Dans ces derniers, toute instruction fournit un résultat.

4.5 EXERCICES

Exercice 4.1. Parmi les déclarations de variables JAVA suivantes, indiquez celles qui sont valides :

```
int i = 0;              short j;              long l1, l2 = 0, l3;
short j = 60000;        int i = 0x10;         char c = 'a';
char c = a;             char c = 0x41;        char c = '\u0041';
boolean b = true;       boolean b = 0;        real r = 0.1;
float f = 0.1;          double d = 0.1;       double d = 0;
float f = 0x10;         double d = .1;        int i = 'a';
```

Exercice 4.2. Trouvez l'erreur présente dans le fragment de programme suivant :

```
final double π;
π = 3.1415926535897931;
final float e = 2.7182818284590451f;
π = e;
```

Exercice 4.3. En fonction des déclarations de variables, indiquez le type de chacune des expressions suivantes :

```
int i, j, k;
double x, y, z;
char c;
boolean b;
```

```
x           2              i = j                    i == j
x + 2.0     x + 2          i + 2                    x + i
x / 2       x / 2.0        i / 2                    i / 2.0
x < y       i % j + y      i / j + y                i > j > k
i && b      i == j && b    i > j && k > j           x + y * i
i = c       x = (int) y    c = (char) (int) c + 1)  i++
```

Exercice 4.4. Écrivez en JAVA les trois expressions suivantes :

$$a^2 - c + \cfrac{a}{bc + \cfrac{c}{d + \cfrac{e}{f}}} \qquad \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad \frac{\frac{1}{a} + \frac{1}{b}}{c + d}$$

Chapitre 5

Énoncés structurés

Les actions que nous avons étudiées jusqu'à présent sont des actions élémentaires. Une action *structurée* est formée à partir d'autres actions, qui peuvent être elles-mêmes élémentaires ou structurées. Dans ce chapitre, nous présenterons deux premières actions structurées, l'*énoncé composé* et l'*énoncé conditionnel*, et pour chacune d'entre elles la règle de déduction qui permet d'en vérifier la validité.

5.1 ÉNONCÉ COMPOSÉ

Comme pour les expressions, il est possible de parenthéser une suite d'actions. L'énoncé composé groupe des actions qui sont ensuite considérées comme une seule action. Dans notre notation algorithmique, les énoncés à composer seront placés entre les deux parenthéseurs **début** et **fin**. Par exemple, la composition de trois énoncés E_1, E_2, E_3 s'écrit de la façon suivante :

début E_1 E_2 E_3 **fin**

Les trois énoncés sont exécutés de façon *séquentielle*, c'est-à-dire successivement les uns *après* les autres. L'énoncé E_2 ne pourra être traité qu'une fois l'exécution de l'énoncé E_1 achevée. De même, l'exécution de E_3 ne commence qu'après la fin de celle de E_2.

La notion d'exécution séquentielle est à mettre en opposition avec celle d'exécution *parallèle*, qui permettrait le traitement *simultané* des trois énoncés. La vérification de la validité des algorithmes parallèles, surtout si les actions s'exécutent de façon synchrone, est beaucoup plus complexe et difficile à mettre en œuvre que celle des algorithmes séquentiels. Les algorithmes présentés dans cet ouvrage sont exclusivement séquentiels.

Règle de déduction

La règle de déduction d'un énoncé composé s'exprime de la façon suivante :

$$\underline{\text{si}}\ \{P\} \Rightarrow \{P_1\} \overset{E_1}{\Rightarrow} \{Q_1\} \Rightarrow \{P_2\} \overset{E_2}{\Rightarrow} \{Q_2\}\ \ldots\ \{P_n\} \overset{E_n}{\Rightarrow} \{Q_n\} \Rightarrow \{Q\}$$
$$\underline{\text{alors}}$$
$$\{P\}\ \textbf{début}\ \{P_1\}\ \mathrm{E}_1\ \{Q_1\}\ \{P_2\}\ \mathrm{E}_2\ \{Q_2\}\ \ldots\ \{P_n\}\ \mathrm{E}_n\ \{Q_n\}\ \textbf{fin}\ \{Q\}$$

La notation $\{P\} \overset{E}{\Rightarrow} \{Q\}$ exprime que le conséquent Q se déduit de l'antécédent P par l'application de l'énoncé E. S'il n'y a pas d'énoncé, la notation $\{P\} \Rightarrow \{Q\}$ indique que Q se déduit directement de P. La règle de déduction précédente spécifie que si la pré-condition

$$\{P\} \Rightarrow \{P_1\} \overset{E_1}{\Rightarrow} \{Q_1\} \Rightarrow \{P_2\} \overset{E_2}{\Rightarrow} \{Q_2\}\ \ldots\ \{P_n\} \overset{E_n}{\Rightarrow} \{Q_n\} \Rightarrow \{Q\}$$

est vérifiée alors le conséquent Q se déduit de l'antécédent P par application de l'énoncé composé.

L'énoncé composé en JAVA

Les parenthéseurs sont représentés par les accolades ouvrantes et fermantes. La plupart des langages de programmation utilise un séparateur entre les énoncés, qui doit être considéré comme un opérateur de séquentialité. En JAVA, il n'y a pas à proprement parlé de séparateur d'énoncé. Toutefois, un point-virgule est nécessaire pour terminer un énoncé simple.

5.2 ÉNONCÉS CONDITIONNELS

Les actions qui forment les programmes que nous avons écrits jusqu'à présent sont exécutées systématiquement une fois. Les langages de programmation proposent des énoncés conditionnels qui permettent d'exécuter ou non une action selon une décision prise en fonction d'un *choix*. Le critère de choix est en général la valeur d'un objet d'un type élémentaire discret.

5.2.1 Énoncé `choix`

Dans la vie courante, nous avons quotidiennement des décisions à prendre, souvent simples, et parfois difficiles. Imaginons un automobiliste abordant une intersection routière contrôlée par un feu de signalisation. Respectueux du code de la route, il sait que si le feu est rouge, il doit s'arrêter ; si le feu est vert, il peut passer ; si le feu est orange, il doit s'arrêter si cela est possible, sinon il passe.

D'un point de vue informatique, il est possible de modéliser le comportement de l'automobiliste à l'aide d'un énoncé **`choix`**. Le critère de choix est la couleur du feu dont le domaine de définition est l'ensemble formé par les trois couleurs rouge, vert et orange. À chacune de ces valeurs est associée une certaine action. Nous pouvons écrire ce choix de la façon suivante :

```
choix couleur du feu parmi
    rouge  : s'arrêter
    vert   : passer
    orange : s'arrêter si possible, sinon passer
finchoix
```

D'une façon plus formelle, l'énoncé **choix** précise l'expression dont l'évaluation fournira la valeur de l'objet discriminatoire, puis il donne la liste des actions possibles, chacune étant précédée de la valeur correspondante. L'énoncé **choix** s'écrit :

```
choix expr parmi
    val1 : E1
    val2 : E2
     ⋮
    valn : En
finchoix
```

L'expression est évaluée et seul l'énoncé qui correspond au résultat obtenu est exécuté. Que se passe-t-il si l'évaluation de l'expression renvoie une valeur non définie dans l'énoncé ? En général, plutôt que de signaler une erreur, les langages choisissent de n'exécuter aucune action.

Règle de déduction

La règle de déduction de l'énoncé **choix** s'exprime de la façon suivante :

si $\forall k \in [1,n]$, $\{P \text{ et } expr = val_k\} \overset{E_k}{\Rightarrow} \{Q_k\}$
alors $\{P\}$ énoncé-choix $\{Q\}$

En pratique, $\{Q\}$ peut être l'union de conséquents $\{Q_k\}$ des énoncés E_k. Notez que le conséquent doit être vérifié, même si aucun énoncé E_k n'a été exécuté.

L'énoncé **choix** en JAVA

La notion de choix est mise en œuvre grâce à l'énoncé **switch**. L'expression, dénotée entre parenthèses, doit rendre une valeur d'un type entier ou caractère. Chaque valeur de la liste des valeurs possibles est introduite par le mot-clé **case**, et il existe une valeur spéciale, appelée **default**. À cette dernière, il est possible d'associer un ou plusieurs énoncés, qui sont exécutés lorsque l'expression renvoie une valeur qui ne fait pas partie de la liste des valeurs énumérées. De plus, le programmeur doit explicitement indiquer, à l'aide de l'instruction **break**, la fin de l'énoncé sélectionné et l'achèvement de l'énoncé **switch**. On peut regretter que le mécanisme de terminaison de l'énoncé **switch** ne soit pas automatique comme le proposent d'autres langages. L'exemple précédent s'écrit en JAVA comme suit :

```
switch (couleurDuFeu) {
    case rouge : s'arrêter; break;
    case vert : passer; break;
    case orange : s'arrêter si possible, sinon passer; break;
}
```

5.2.2 Énoncé `si`

Lorsque l'énoncé **choix** est gouverné par la valeur d'un prédicat binaire, c'est-à-dire une expression booléenne, comme par exemple, « si mon salaire augmente alors je reste sinon je change d'entreprise », on est dans un cas particulier de l'énoncé **choix**, appelé l'énoncé **si**. Au lieu d'écrire :

```
choix mon salaire augmente parmi
    vrai  : je reste
    faux  : je change d'entreprise
finchoix
```

on préférera la formule suivante, plus proche du langage courant :

```
si mon salaire augmente
    alors je reste sinon je change d'entreprise
finsi
```

D'une façon générale, cet énoncé conditionnel s'exprime de la façon suivante :

si B **alors** E_1 **sinon** E_2 **finsi**

L'énoncé E_1 est exécuté si le prédicat booléen B est vrai, sinon ce sera l'énoncé E_2 qui le sera.

Règle de déduction

La règle de déduction de l'énoncé **si** est donnée ci-dessous :

$$\underline{\text{si}}\ \{P\ et\ B\} \overset{E_1}{\Rightarrow} \{Q_1\}\ \underline{\text{et}}\ \{P\ et\ non\ B\} \overset{E_2}{\Rightarrow} \{Q_2\}$$
$$\underline{\text{alors}}\ \{P\}\ \text{énoncé-si}\ \{Q\}$$

L'exécution de l'énoncé E_1 ou celle de E_2 doit conduire au conséquent $\{Q\}$. Notez que ce dernier peut être l'union de deux conséquents particuliers $\{Q_1\}$ et $\{Q_2\}$ des énoncés E_1 et E_2.

Forme réduite

Il est fréquent que l'action à effectuer dans le cas où le prédicat booléen est faux soit vide. La partie « sinon » est alors omise et on obtient une forme réduite de l'énoncé **si**. D'une façon générale, cette forme s'écrit :

si B **alors** E **finsi**

Par exemple, pour obtenir la valeur absolue d'un entier, nous écrirons l'énoncé suivant :

```
{x est un entier quelconque}
si x < 0 alors x ← -x finsi
{x ⩾ 0}
```

Règle de déduction

La règle de déduction de la forme réduite de l'énoncé **si** s'exprime de la façon suivante :

$$\underline{\texttt{si}}\ \{P\ et\ B\} \overset{E}{\Rightarrow} \{Q\}\ \underline{\texttt{et}}\ \{P\ et\ non\ B\} \Rightarrow \{Q\}$$
`alors` {P} énoncé-si-réduit {Q}

Notez que si l'énoncé `E` n'est pas exécuté, le conséquent $\{Q\}$ doit être vérifié.

L'énoncé **si** en JAVA

L'énoncé **si** et sa forme réduite s'écrivent en JAVA de la façon suivante :

```
if (B) E1 else E2
if (B) E
```

Notez que cette syntaxe ne s'embarrasse pas des mots-clés **alors** et **finsi** mais au détriment d'une certaine lisibilité, et a pour conséquence l'obligation de mettre systématiquement le prédicat booléen entre parenthèses.

5.3 RÉSOLUTION D'UNE ÉQUATION DU SECOND DEGRÉ

Avec l'énoncé conditionnel, il nous est désormais possible d'écrire des programmes plus conséquents. La programmation de la résolution d'une équation du second degré a ceci d'intéressant qu'elle est d'une taille suffisamment importante (mais pas trop) pour mettre en évidence, d'une part, la construction progressive d'un algorithme, et d'autre part, l'influence de l'inexactitude de l'arithmétique réelle sur l'algorithme.

Posons le problème. On désire écrire un programme qui calcule les racines d'une équation non dégénérée du second degré. Formalisons un peu cet énoncé : soient `a`, `b` et `c`, trois coefficients réels d'une équation du second degré avec `a`$\neq$`0`, calculer les racines `r1`+$i\times$`i1` et `r2`+$i\times$`i2` solutions de l'équation.

La suite d'actions qui assure ce travail peut être réduite à une action élémentaire, qui a pour données les trois coefficients `a`, `b` et `c` et pour résultats les parties réelles (`r1` et `r2`) et imaginaires (`i1` et `i2`) des deux racines. Ce que nous écrivons :

```
{Antécédent : a≠0, b et c réels coefficients de
              l'équation du second degré, ax²+bx+c}
calculer les racines de l'équation
{Conséquent : (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
```

S'il existe une procédure prédéfinie qui calcule les racines et qui respecte le conséquent final, alors il suffit de l'appeler et notre travail est terminé. Dans le cas contraire, nous devons procéder au calcul. Un premier niveau de réflexion peut être le suivant. Regardons la valeur du discriminant Δ, si $\Delta \geqslant 0$, les racines sont réelles, sinon elles sont complexes. Cela se traduit par l'algorithme :

```
Algorithme Équation du second degré
   {Antécédent : a≠0, b et c réels coefficients de
                 l'équation du second degré, ax²+bx+c}
   calculer le discriminant Δ
   si Δ⩾0 alors calculer les racines réelles
      sinon calculer les racines complexes
   finsi
   {Conséquent : (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
```

Dans une seconde étape, il est nécessaire de détailler les trois parties énoncées de façon informelle et qui se dégagent de l'algorithme précédent. Notez que ces trois parties sont indépendantes, et qu'elles peuvent être traitées dans un ordre quelconque.

1. Le calcul du discriminant s'écrit directement. C'est une simple expression mathématique : Δ←carré(b)-4×a×c.
2. Pour le calcul des racines réelles, il est absolument nécessaire de tenir compte du fait que nous travaillons sur des objets de type réel, et que l'arithmétique réelle sur les ordinateurs est inexacte. Le calcul direct des racines selon les formules mathématiques habituelles est dangereux, en particulier, si l'opération d'addition ou de soustraction conduit à soustraire des valeurs presque égales. Pour éviter cette situation, on calcule d'abord la racine la plus grande en valeur absolue. Si b>0, alors on calcule $(-b-\sqrt{\Delta})/2a$, sinon c'est $(-b+\sqrt{\Delta})/2a$. Une fois ce calcul effectué, la seconde racine se déduit de la première par le produit $r_1 \times r2 = c/a$, sauf si la première est nulle. Auquel cas, la seconde l'est aussi. Enfin, dans le cas réel, les deux parties imaginaires sont nulles.
   Les opérations de comparaison sont elles aussi dangereuses, en particulier l'opération d'égalité. La comparaison d'un nombre avec zéro devra se faire à un epsilon près. En tenant compte de tout ce qui vient d'être écrit, l'algorithme du calcul des racines réelles s'écrit de la façon suivante :

```
{calcul des racines réelles}
si b>0 alors r1 ← -(b+√Δ)/(2×a)
       sinon r1 ← (√Δ-b)/(2×a)
finsi
{r1 est la racine la plus grande en valeur absolue}
si |r1| < ε alors r2 ←0
  sinon r2 ← c/(a×r1)
finsi
i1 ← 0
i2 ← 0
{(x-r1)(x-r2)=0}
```

3. Le calcul des racines complexes ne pose pas de problème particulier. Les racines complexes sont données par les expressions suivantes :

```
r1 ← r2 ← -b/(2×a)
i1 ← √(-Δ)/(2×a)
i2 ← -i1
```

En rassemblant les différentes parties, l'algorithme complet de résolution d'une équation du second degré non dégénérée est le suivant :

```
Algorithme Équation du second degré
    {Antécédent : a≠0, b et c réels coefficients de
                  l'équation du second degré, ax²+bx+c}
    {Conséquent : (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
    constante
        ε = ? {dépend de la précision des réels sur la machine}
    variables
        Δ, a, b, c, r1, r2, i1, i2 type réel
        {a≠0, b et c coefficients réels de
         l'équation du second degré, ax²+bx+c}
        Δ ←carré(b)-4×a×c
        si Δ≠0 alors {calcul des racines réelles}
            si b>0 alors r1 ← -(b+√Δ)/(2×a)
               sinon r1 ← (√Δ-b)/(2×a)
            finsi
            {r1 est la racine la plus grande en valeur absolue}
            si |r1|< ε alors r2 ←0
                       sinon r2 ← c/(a×r1)
            finsi
            i1 ← 0
            i2 ← 0
            {(x-r1)(x-r2)=0}
        sinon  {calcul des racines complexes}
            r1 ← r2 ← -b/(2×a)
            i1 ← √(-Δ)/(2×a)
            i2 ← -i1
        finsi
        {(x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
```

L'écriture en JAVA du programme est maintenant aisée, il s'agit d'une simple transcription de l'algorithme précédent. Insistons encore sur le fait que la tâche la plus difficile, lors de la construction d'un programme, est la conception de l'algorithme et non pas son écriture dans un langage particulier.

```
/** La classe Éq2Degré résout une équation du second degré
 *  non dégénérée, à partir de ses trois coefficients lus sur
 *  l'entrée standard. Elle affiche sur la sortie standard les
 *  racines solutions de l'équation.
 */
import java.io.*;
class Éq2Degré {
    public static void main (String[] args) throws IOException
    {
        final double ε = 01e-100; //précision du calcul
        double a = StdInput.readDouble(),
               b = StdInput.readDouble(),
               c = StdInput.readlnDouble(),
```

```
            r1, r2, i1, i2, Δ;
        // a<>0, b et c coefficients réels de
        // l'équation du second degré, ax²+bx+c
        Δ = (b*b)-4*a*c;
        if (Δ>=0) { //calcul des racines réelles
            if (b>0) r1 = -(b+Math.sqrt(Δ))/(2*a);
            else r1 = (Math.sqrt(Δ)-b)/(2*a);
            // r1 est la racine la plus grande en valeur absolue
            r2 =  Math.abs(r1) < ε ? 0 : c/(a*r1);
            i1 = i2 = 0;
            // (x-r1)(x-r2)=0
        }
        else { //calcul des racines complexes
            r1 = r2 = -b/(2*a);
            i1 = Math.sqrt(-Δ)/(2*a);
            i2 = -i1;
        }
        // (x-(r1+i×i1)) (x-(r2+i×i2)) = 0
        // écrire les racines solutions sur la sortie standard
        System.out.println("r1=␣(" + r1 + "," + i1 + ")");
        System.out.println("r2=␣(" + r2 + "," + i2 + ")");
    }
} //fin classe Éq2Degré
```

Les noms de constantes et de variables ε et Δ correspondent aux caractères UNICODE `\u03b5` et `\u0394`. Remarquez que ce programme emploie une nouvelle construction du langage JAVA, l'expression conditionnelle (le seul opérateur ternaire du langage). Celle-ci a la forme suivante :

```
exp1 ? exp2 : exp3
```

L'expression booléenne `exp1` est évaluée. Si le résultat est vrai, le résultat de l'expression conditionnelle est le résultat de l'évaluation de l'expression `exp2` ; sinon le résultat est celui de l'évaluation de `exp3`. Notez que le calcul de `r2` aurait tout aussi bien pu s'écrire :

```
if (Math.abs(r1) < ε)) r2 = 0;
  else r2 = c/(a*r1);
```

5.4 EXERCICES

Exercice 5.1. Écrivez l'algorithme de l'addition de deux entiers `x` et `y` qui vérifie qu'elle ne provoque pas un dépassement de capacité. Vous prouverez la validité de votre algorithme en appliquant les règles de déduction de l'énoncé conditionnel **`si`**.

Exercice 5.2. Écrivez un programme qui calcule les deux racines réelles d'une équation du second degré, à partir des formules mathématiques classiques, sans tenir compte de l'inexactitude de l'arithmétique réelle. Cherchez des valeurs pour les coefficients `a`, `b` et `c` qui montrent que la méthode présentée dans ce chapitre fournit des résultats plus satisfaisants.

Chapitre 6

Procédures et fonctions

Dans une approche de la programmation organisée autour des actions, les programmes sont d'abord structurés à l'aide de procédures et de fonctions. À l'instar de C ou PASCAL, les langages qui suivent cette approche sont dits *procéduraux*. Nous avons vu précédemment comment utiliser une procédure et une fonction prédéfinie ; dans ce chapitre, nous étudierons comment les construire et le mécanisme de transmission des paramètres lorsqu'elles sont appelées.

6.1 INTÉRÊT

Il arrive fréquemment qu'une même suite d'énoncés doive être exécutée en plusieurs points du programme. Une *procédure* ou une *fonction* permet d'associer un *nom* à une suite d'énoncés, et d'utiliser ce nom comme abréviation chaque fois que la suite apparaît dans le programme. Bien souvent, dans la terminologie des langages procéduraux, le terme *sous-programme* est utilisé pour désigner indifféremment une procédure ou une fonction.

L'association d'un nom à la suite d'énoncés se fait par une *déclaration* de sous-programme. L'utilisation du nom à la place de la suite d'énoncés s'appelle un *appel* de procédure ou de fonction.

Si la plupart des langages de programmation possèdent la notion de procédures et de fonctions, c'est bien parce qu'elle est un outil fondamental de « *l'art de la programmation dont la maîtrise influe de façon décisive sur le style et la qualité du travail du programmeur* » [Wir75].

Les sous-programmes ont un rôle très important dans la *structuration* et la *localisation*, le *paramétrage* et la *lisibilité* des programmes :

- Bien plus qu'une façon d'abréger le texte du programme, c'est un véritable outil de *structuration* des programmes en composants *fermés* et *cohérents*. Nous avons vu dans l'introduction, que la programmation descendante par raffinements successifs divisait le problème en sous-parties cohérentes. Les sous-programmes seront naturellement les outils adaptés à la description de ces sous-parties. Une suite d'énoncés pourra être déclarée comme sous-programme même si celle-ci n'est exécutée qu'une seule fois.
- Il arrive fréquemment que, pour des besoins de calculs intermédiaires, une suite d'énoncés ait besoin de définir des variables (ou plus généralement des objets) qui sont sans signification en dehors de cette suite, comme par exemple la variable Δ dans le calcul des racines d'une équation du second degré. Les procédures ou fonctions seront des *unités textuelles* permettant de définir des *objets locaux* dont le *domaine de validité* sera bien délimité.
- Certaines suites d'énoncés ont de fortes ressemblances, mais ne diffèrent que par la valeur de certains identificateurs ou expressions. On aimerait que la suite d'énoncés, qui calcule les racines de l'équation du second degré donnée au chapitre précédent, puisse faire ce calcul avec des coefficients chaque fois différents. Le mécanisme de *paramétrage* d'une fonction ou d'une procédure nous permettra de considérer la suite d'énoncés comme « un schéma de calcul abstrait[1] » dont les *paramètres* représenteront des valeurs particulières à chaque exécution particulière de la suite d'énoncés.
- Une conséquence de l'effet de structuration des programmes à l'aide des sous-programmes est l'augmentation de la lisibilité du code. De plus, les procédures et fonctions amélioreront la documentation des programmes.

6.2 DÉCLARATION D'UN SOUS-PROGRAMME

Le rôle de la déclaration d'un sous-programme est de lier un nom unique à une suite d'énoncés sur des objets *formels* ne prenant des valeurs *effectives* qu'au moment de l'appel de ce sous-programme. Le nom est un identificateur de procédure ou de fonction. Cette déclaration est toujours formée d'un *en-tête* et d'un *corps*.

L'en-tête

L'en-tête du sous-programme, appelé aussi *signature* ou encore *prototype*, spécifie :

- le *nom* du sous-programme ;
- les *paramètres formels* et leur type ;
- le *type du résultat* dans le cas d'une fonction.

La déclaration d'une procédure prend la forme suivante :

```
procédure NomProc ([ <liste de paramètres formels> ])
{Antécédent: une affirmation}
{Conséquent: une affirmation}
{Rôle: une affirmation donnant le rôle de la procédure}
```

1. *ibidem.*

Notez que les crochets indiquent que la liste des paramètres formels est facultative. Tous les en-têtes de procédures doivent contenir des affirmations décrivant leur antécédent, leur conséquent ou leur rôle. L'en-tête correspondant à la déclaration de la procédure qui calcule les racines d'une équation du second degré peut être :

```
procédure Éq2degré(données a, b, c : réel
                   résultats r1, i1, r2, i2 : réel)
{Antécédent : a≠0, b et c réels coefficients de
              l'équation du second degré, ax²+bx+c}
{Conséquent : (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
{Rôle       : calcule les racines d'une équation du second degré}
```

La procédure s'appelle Éq2degré, possède sept paramètres formels a, b, c qui sont les données de la procédure, et r1, i1, r2 et i2 qui sont les résultats qu'elle calcule. Tous ces paramètres sont de type réel.

Une fonction est un sous-programme qui représente une valeur résultat qui peut intervenir dans une expression. La déclaration d'une fonction précise en plus le type du résultat renvoyé :

```
fonction NomFonc ([ <liste de paramètres formels> ]) : type-résultat
{Antécédent : une affirmation}
{Conséquent : une affirmation}
{Rôle       : une affirmation donnant le rôle de la fonction}
```

L'en-tête suivant déclare une fonction qui retourne la racine carrée d'un entier naturel :

```
fonction rac2(x: naturel): réel
{Antécédent : x ⩾ 0}
{Conséquent : rac2 = √x}
{Rôle       : calcule la racine carrée de l'entier naturel x}
```

Le corps

Le corps du sous-programme contient la suite d'énoncés. Nous le délimiterons par les mots-clés **finproc** ou **finfonc**, et il se place à la suite de l'en-tête du sous-programme.

```
{corps de la procédure Éq2degré}
   ... suite d'énoncés ...
finproc {Éq2degré}


{corps de la fonction rac2}
   ... suite d'énoncés ...
finfonc {rac2}
```

6.3 APPEL D'UN SOUS-PROGRAMME

L'*appel* d'un sous-programme est une action élémentaire qui permet l'exécution de la suite d'énoncés associée à son nom. Il consiste simplement à indiquer le nom de la procédure ou de la fonction avec ses *paramètres effectifs*.

```
{appel de la procédure Éq2degré}
Éq2degré(u,1,v+1,rl1,ig1,rl2,ig2)
{deux appels de la fonction rac2}
écrire(rac2(5), rac2(tangente))
```

6.4 TRANSMISSION DES PARAMÈTRES

Un paramètre formel est un nom sous lequel un paramètre d'un sous-programme est connu à l'intérieur de celui-ci. Un paramètre effectif est l'entité fournie au moment de l'appel du sous-programme, sous la forme d'un nom ou d'une expression.

Nous distinguerons deux types de paramètres formels : les *données* et les *résultats*. Les paramètres données fournissent les valeurs à partir desquelles les énoncés du corps du sous-programme effectueront leur calcul. Les paramètres résultats donnent les valeurs calculées par la procédure. Une procédure peut avoir un nombre quelconque de paramètres données ou résultats. En revanche, une fonction, puisqu'elle renvoie un résultat unique, n'aura que des paramètres données. Dans bien des langages de programmation, rien n'interdit de déclarer dans l'en-tête d'une fonction des paramètres résultats, mais nous considérerons que c'est une faute de programmation.

Le remplacement des paramètres formels par les paramètres effectifs au moment de l'appel du sous-programme se fait selon des règles strictes de *transmission* des paramètres. Pour de nombreux des langages de programmation, le nombre de paramètres effectifs doit être identique à celui des paramètres formels, et la correspondance entre les paramètres formels et effectifs se fait sur la position. De plus, ils doivent être de type compatible (*e.g.* x ne pourrait pas être de type booléen). Nous utiliserons cette convention.

```
procédure P(données a, b : entier
            résultat c : entier)
{Antécédent : ... }
{Conséquent : ... }
{Rôle       : ... }
{début du corps de P}
...
{finproc {P}

{appel de P}
P(x,y,z)
```

Les paramètres effectifs x, y et z correspondent, respectivement, aux paramètres formels a, b et c.

Parmi tous les modes de transmission de paramètres que les langages définissent (et il en existe de nombreux), nous allons en présenter deux : la transmission par valeur et la transmission par référence.

6.4.1 Transmission par valeur

La transmission par valeur est utilisée pour les paramètres données. Elle a pour effet d'affecter au nom du paramètre formel la *valeur* du résultat de l'évaluation du paramètre effectif. Le paramètre effectif sert à fournir une valeur initiale au paramètre formel. Dans l'appel `P(`x_e`)` d'une procédure `P` dont l'en-tête est le suivant :

```
procédure P(donnée xf : T)
```

tout se passe comme si, avant d'exécuter la suite d'énoncés de la procédure `P`, l'affectation $x_f \leftarrow x_e$ était effectuée.

Notez que le paramètre effectif peut donc être un nom ou une expression. D'autre part, toute modification du paramètre formel reste locale au sous-programme, cela veut dire qu'un paramètre effectif transmis par valeur ne peut être modifié.

6.4.2 Transmission par résultat

De quelle façon un sous-programme retourne-t-il ses résultats ? Une procédure utilisera le mode de transmission par *résultat*. Ce mode de transmission est précisé en faisant précéder le nom du paramètre formel par le mot-clé **`résultat`**. Dans l'appel `P(`x_e`)` d'une procédure `P` dont l'en-tête est le suivant :

```
procédure P(résultat xf : T)
```

tout se passe comme si, en fin d'exécution de la procédure `P`, le paramètre effectif x_e était modifié par l'affectation $x_e \leftarrow x_f$.

Les paramètres effectifs transmis par résultat sont nécessairement des *noms*, puisqu'ils apparaissent en partie droite de l'affectation, et en aucun cas des expressions.

Notez qu'une fonction ne devra comporter aucun paramètre transmis par résultat, puisque par définition elle retourne un seul résultat dénoté par l'appel de fonction.

D'autre part, le mode de transmission d'un paramètre qui est à la fois donnée et résultat est le mode par résultat. Les règles de transmission précédentes sont appliquées à l'entrée et à la sortie du sous-programme.

6.5 RETOUR D'UN SOUS-PROGRAMME

L'endroit où se fait l'appel du sous-programme s'appelle le *contexte d'appel*. Ce contexte peut être soit le programme principal, soit un autre sous-programme. Après l'appel, les énoncés du sous-programme appelé sont exécutés. À l'issue de leur exécution, le sous-programme s'achève et le prochain énoncé exécuté est celui qui suit immédiatement l'énoncé d'appel du sous-programme dans le contexte d'appel.

Nous avons vu que l'appel d'une fonction délivrait un résultat. Comment est spécifié ce résultat dans la fonction appelée ? Il le sera au moyen de l'instruction **`rendre`** `expr`, où `expr` est une expression compatible avec le type indiqué dans l'en-tête de la déclaration de la fonction.

```
fonction F([<paramètres formels>]) : T
    ...
    rendre expr {expr délivre un résultat de type T}
    ...
finfonc {F}
```

Le corps de la fonction peut contenir plusieurs instructions **rendre**, mais l'exécution du premier **rendre** a pour effet d'achever l'exécution de la fonction, et de revenir au contexte d'appel avec la valeur résultat.

6.6 LOCALISATION

À l'intérieur d'un sous-programme, il est possible de déclarer des variables ou des constantes pour désigner de nouveaux objets. Ces objets sont dits *locaux*. Chaque fois que des objets n'ont de signification qu'à l'intérieur d'un sous-programme, ils devront être déclarés locaux au sous-programme. Les noms locaux doivent être déclarés entre l'en-tête du sous-programme et son corps. Dans la procédure Éq2degré, la variable Δ et la constante ε doivent être déclarées locales à cette procédure puisqu'elles ne sont utilisées que dans cette procédure.

```
procédure Éq2degré(données a, b, c : réel ;
                   résultats r1, i1, r2, i2 : réel)
{Antécédent : a≠0, b et c réels coefficients de
              l'équation du second degré, ax²+bx+c}
{Conséquent : (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
{Rôle: calcule les racines d'une équation du second degré}
constante ε = ? {dépend de la précision des réels sur la machine}
variable Δ type réel
    ...
finproc {Éq2degré}
```

La validité des objets locaux est le texte du sous-programme tout entier. Les objets locaux, en particulier les variables, démarrent leur existence au moment de l'appel du sous-programme, et sont détruites lors de l'achèvement du sous-programme. Leur *durée de vie* est égale à celle de l'exécution du sous-programme dans lequel ils sont définis.

Les objets qui sont déclarés à l'intérieur du sous-programme sont dits *non locaux*. S'ils sont définis en dehors, ils sont dits *globaux*, et s'ils sont prédéfinis par le langage, ils sont globaux et *standard*. Les objets globaux et standard ont une durée de vie égale à celle du programme tout entier.

Un autre aspect de la validité des objets locaux concerne l'utilisation de leur nom, ce qu'on appelle la *visibilité* des noms. Un nom défini dans un sous-programme est visible partout à l'intérieur du sous-programme et invisible à l'extérieur. Ainsi la constante ε et la variable Δ sont visibles partout dans la procédure Éq2degré et invisibles, *i.e.* inaccessibles à l'extérieur. Les objets globaux et standard sont visibles en tout point du programme.

Certains langages autorisent la déclaration de sous-programmes locaux à d'autres sous-programmes. Par exemple, on peut déclarer une procédure locale à une autre pour effectuer

un calcul particulier, et qui n'a de sens que dans cette procédure. On parle alors de sous-programmes emboîtés.

```
procédure P1
{Antécédent: ...}
{Conséquent: ...}
{Rôle: ...}

   procédure P2
   {Antécédent: ...}
   {Conséquent: ...}
   {Rôle: ...}

   {corps de P2}
       ...
   finproc {P2}

{corps de P1}
    P2
finproc {P1}
```

Nous venons de dire qu'un nom défini dans un sous-programme est visible partout à l'intérieur du sous-programme. Cette affirmation doit être modulée. Imaginons que la procédure P1 précédente possède une variable locale x. Est-il possible ou non de déclarer dans P2 une variable (et de façon générale un nom d'objet) du même nom que x? La réponse est oui.

```
procédure P1
{Antécédent : ...}
{Conséquent : ...}
{Rôle: ...}

variable x type T

   procédure P2
   {Antécédent : ...}
   {Conséquent : ...}
   {Rôle: ...}

   constante x = 1234
   {procédure locale à P1}
          ... x ... ⇐ {la constante x de P2}
   finproc {P2}

{corps de P1}
    P2
    ... x ... ⇐ {la variable x de P1}
finproc {P1}
```

On peut alors préciser notre règle : un nom défini dans un sous-programme est uniquement visible à l'intérieur du sous-programme. Cette visibilité peut être limitée par tout homonyme déclaré dans un sous-programme emboîté.

À l'intérieur d'un sous-programme, on peut donc manipuler des objets locaux et non locaux, ainsi que les paramètres. La manipulation directe des objets non locaux, en particulier globaux, qui appartiennent à l'environnement du sous-programme, est considérée comme dangereuse puisqu'elle peut se faire depuis n'importe quel point du programme sans contrôle. Une telle action s'appelle un *effet de bord* et devra être évitée.

➢ Règle

Nous adopterons la discipline suivante : on communique avec la procédure en entrée grâce aux paramètres données et en sortie grâce aux paramètres résultats. Les énoncés, à l'intérieur du sous-programme, ne manipuleront que des objets locaux ou des paramètres formels.

Cette règle correspond bien à la vision d'un sous-programme : une boîte noire qui calcule, à partir de données, des résultats. L'interface entre le contexte d'appel du sous-programme et l'intérieur du sous-programme est donnée par l'en-tête du sous-programme, c'est-à-dire les paramètres données et résultats.

6.7 RÈGLES DE DÉDUCTION

Les règles de déduction pour un sous-programme seront bien sûr fonction des énoncés particuliers contenus dans son corps. Mais on peut dire que pour :

```
procédure P(donnée pf1 : T1
            résultat pf2 : T2)
{Antécédent : A = affirmation sur pf1}
{Conséquent : C = affirmation sur pf1 et pf2}
```

L'antécédent `A` de `P` est une affirmation sur la donnée `pf1`. Le conséquent `C` de `P` est une affirmation sur la donnée `pf1` et le résultat `pf2`.

➢ Règle 1

D'une façon générale, l'antécédent d'un sous-programme est une affirmation sur l'ensemble des paramètres formels données (et ceux à la fois données et résultats, s'ils existent). Le conséquent est une affirmation sur l'ensemble des paramètres données et des paramètres résultats (et ceux à la fois données et résultats s'ils existent).

➢ Règle 2

Les deux affirmations `A` et `C` ne doivent contenir aucune référence aux paramètres effectifs.

➢ Règle 3

Aucun objet local ne doit apparaître dans l'antécédent ou le conséquent.

On peut constater que ces trois règles sont bien vérifiées dans l'antécédent et le conséquent de la procédure `Éq2degré`.

```
procédure Éq2degré(données a, b, c : réel
                   résultat r1, i1, r2, i2: réel)
{Antécédent : a≠0, b et c réels coefficients de
              l'équation du second degré, ax²+bx+c}
{Conséquent : (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
```

Lors d'un appel de la procédure `P` avec les paramètres effectifs `pe1` et `pe2` :

$\{A^{pe1}_{pf1}\}$ `P(pe1,pe2)` $\{C^{pe1,pe2}_{pf1,fp2}\}$

L'antécédent de l'appel de la procédure `P` est l'affirmation `A` dans laquelle `pf1` a été remplacé par `pe1` ; et son conséquent est l'affirmation `C` dans laquelle `pf1` et `pf2` ont été remplacés, respectivement, par `pe1` et `pe2`.

Règle 4

D'une façon générale, l'antécédent de l'appel d'un sous-programme est l'antécédent du sous-programme dans lequel l'ensemble des paramètres formels données (et ceux à la fois données et résultats, s'ils existent) ont été remplacés par les paramètres effectifs données (et ceux à la fois données et résultats, s'ils existent). Le conséquent de l'appel est le conséquent du sous-programme dans lequel l'ensemble des paramètres formels données et résultats (et ceux à la fois données et résultats, s'ils existent) ont été remplacés par les paramètres effectifs données et résultats (et ceux à la fois données et résultats, s'ils existent).

Règle 5

Après remplacement des paramètres formels par les paramètres effectifs, les affirmations ne doivent plus contenir de références à des paramètres formels.

En appliquant les règles précédentes, l'antécédent et le conséquent de l'appel de la procédure `Éq2degré` de la page 49 sont :

```
{Antécédent : u≠0, 1 et v+1 réels coefficients de
              l'équation du second degré, ux²+x+v+1}
Éq2degré(u,1,v+1,rl1,ig1,rl2,ig2)
{Conséquent : (x-(rl1+i×ig1)) (x-(rl2+i×ig2)) = 0}
```

6.8 EXEMPLES

L'écriture de l'algorithme de la page 45 dans lequel le calcul des racines est fait par une procédure est réécrit de la façon suivante :

```
Algorithme Équation second degré
    {Antécédent : coeff1≠0, coeff2 et coeff3 réels coefficients
                  d'une équation du second degré}
    {Conséquent : (x-(préel+i×imag1)) (x-(préel2+i×imag2)) = 0}
     variables
            {les trois coefficients de l'équation}
            coeff1, coeff2, coeff3,
           {les deux racines}
            préel1, préel2, pimag1, pimag2 type réel
```

```
{lire les 3 coefficients de l'équation}
lire(coeff1,coeff2,coeff3)
{coeff1, coeff2, coeff3 coefficients réels de l'équation
          coeff3x²+coeff2x+coeff3=0 et coeff1≠0}
Éq2degré(coeff1,coeff2,coeff3,préel1,pimag1,préel2,pimag2)
{(x-(préel1+i×pimag1)) (x-(préel2+i×pimag2)) = 0}
{écrire les résultats}
écrire(préel1,pimag1,préel2,pimag2)


procédure Éq2degré(données a, b, c : réel
                   résultats r1, i1, r2, i2: réel)
{Antécédent: a, b, c coefficients réels de l'équation
             ax²+bx+c=0 avec a≠0}
{Conséquent: (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
{Rôle: calcule les racines d'une équation du second degré}
constante ε = ? {dépend de la précision des réels sur la machine}
variable Δ type réel {le discriminant}

    Δ ←carré(b)-4×a×c
    si Δ≠0 alors {calcul des racines réelles}
       si b>0 alors r1 ← -(b+√Δ)/(2×a)
          sinon r1 ← (√Δ-b)/(2×a)
       finsi
       {r1 est la racine la plus grande en valeur absolue}
       si |r1|< ε alors r2 ←0
                  sinon r2 ← c/(a×r1)
       finsi
       i1 ← 0
       i2 ← 0
       {(x-r1)(x-r2)=0}
    sinon {calcul des racines complexes}
       r1 ← r2 ← -b/(2×a)
       i1 ← √(-Δ)/(2×a)
       i2 ← -i1
    finsi
    {(x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
finproc {Éq2degré}
```

On désire maintenant écrire un programme qui affiche la date du lendemain, avec le mois en toutes lettres. La date du jour est représentée par trois entiers : jour, mois et année. Le calcul de la date du lendemain ne pourra se faire que sur une date *valide* dont l'année est postérieure à 1582[2].

2. Date de l'adoption du calendrier grégorien.

```
Algorithme Date du lendemain
   {Antécédent : trois entiers qui représente une date}
   {Conséquent : la date du lendemain, si elle existe, est affichée}
     lire(jour, mois, année)
     vérifier si la date est valide
     si non valide alors
        signaler l'erreur
     sinon
         calculer la date du lendemain
         afficher la date du lendemain
     finsi
```

Le calcul de la date du lendemain nécessite la connaissance du nombre de jours maximum dans le mois. La vérification de la date déterminera cette valeur. La vérification de la date sera représentée par la procédure `valide` dont l'en-tête est le suivant :

```
procédure valide(données j,m,a: entier
                 résultats çava: booléen
                           max: entier)
{Antécédent: j,m,a trois entiers quelconques qui représentent
             une date jour/mois/année valide ou non }
{Conséquent: date valide ⇒ çava=vrai et max=nombre de jours max du mois
             date non valide ⇒ çava=faux et max est indéterminé}
```

L'algorithme principal avec les déclarations de variables s'écrit alors :

```
Algorithme Date du lendemain
   {Rôle : lit sur l'entrée standard une date donnée sous la
    forme de trois entiers correspondant au jour, au mois
    et à l'année,  et écrit sur la sortie standard la date du
    lendemain si elle existe. L'année doit être supérieure à une
    certaine date minimum et le jour doit correspondre à un jour
    existant. Le mois est écrit en toutes lettres.}
   constante
      annéeMin = 1582
   variables
      jour, mois, année, nbreJours type entier
      ok type booléen

   lire(jour, mois, année)
   valide(jour, mois, année, ok, nbreJours)
   si non ok alors
       écrire("la date donnée n'a pas de lendemain")
   sinon
       {la date est bonne on cherche son lendemain}
       {jour ⩽ nbreJours et mois dans [1,12]}
       si jour < nbreJours alors
            {le mois et l'année de changent pas}
            jour←jour+1
         sinon
            {c'est le dernier jour du mois, il faut passer au
```

```
                premier jour du mois suivant}
               jour←1
               si mois<12 alors mois←mois+1
               sinon
                  {c'est le dernier mois de l'année, il faut passer
                  au premier mois de l'année suivante}
                  mois←1 année←année+1
               finsi
            finsi
            {écrire la date du lendemain}
            écrireDate(jour, mois, année)
         finsi
```

Rappelons qu'une année est bissextile si elle est divisible par 4, mais pas par 100, ou bien si elle est divisible par 400. L'année 1900 n'est pas bissextile alors que 2000 l'est. Notez que la fonction `bissextile` est locale à la procédure `valide`. Elle n'a de signification que dans cette procédure. La procédure `valide` s'écrit :

```
procédure valide(données j, m, a: entier
                 résultats çava: booléen
                           max: entier)
{Antécédent: j,m,a trois entiers quelconques qui représentent
             une date jour/mois/année valide ou non }
{Conséquent: date valide ⇒ çava=vrai et max=nombre de jours max du mois
             date non valide ⇒ çava=faux et max est indéterminé}
    fonction bissextile(donnée année: entier): booléen
    {Antécédent: année≥1600}
    {Conséquent: bissextile = vrai si l'année est bissextile
                              faux sinon}
       rendre (année modulo 4 = 0 et année modulo 100 ≠ 0)
                 ou (année modulo 400 = 0)
    finfonc {bissextile}

{corps de la procédure valide}
    si a < annéeMin alors çava←faux
    sinon {l'année est bonne}
         si (m<1) ou (m>12) alors çava←faux
         sinon {le mois est bon, tester le jour}
            choix m parmi
              1, 3, 5, 7, 8, 10, 12 : max←31
              4, 6, 9, 11 : max←30
              2 :  si bissextile(a) alors max←29
                     sinon max←28
                   finsi
            finchoix
            {max = nombre du jour dans le mois m}
            çava←(j≥1) et (j≤max)
         finsi
    finsi
finproc {valide}
```

Enfin, écrivons la procédure `écrireDate` :

```
procédure écrireDate(données j, m, a : entier)
{Antécédent: j, m, a représentent une date valide}
{Conséquent: la date est écrite sur la sortie standard
             avec le mois en toutes lettres}
   écrire(j, '␣')
   choix m parmi
     1  : écrire("janvier")
     2  : écrire("février")
     3  : écrire("mars")
     4  : écrire("avril")
     5  : écrire("mai")
     6  : écrire("juin")
     7  : écrire("juillet")
     8  : écrire("août")
     9  : écrire("septembre")
     10 : écrire("octobre")
     11 : écrire("novembre")
     12 : écrire("décembre")
   finchoix
   écrire('␣', a)
finproc {écrireDate}
```

Chapitre 7

Programmation par objets

La programmation par objets est une méthodologie qui fonde la structure des programmes autour des objets. Dans ce chapitre, nous présenterons les éléments de base d'un langage de classe : objets, classes, instances et méthodes.

7.1 OBJETS ET CLASSES

Nous avons vu qu'un objet élémentaire n'est accessible que dans sa totalité. Un objet structuré est, par opposition, construit à partir d'un ensemble fini de composants, accessibles indépendamment les uns des autres.

Dans le monde de la programmation par objets, un programme est un système d'interaction d'une collection d'*objets dynamiques*. Selon B. MEYER [Mey88, Mey97], chaque objet peut être considéré comme un *fournisseur* de services utilisés par d'autres objets, les *clients*. Notez que chaque objet peut être à la fois fournisseur et client. Un programme est ainsi vu comme un ensemble de relations contractuelles entre fournisseurs de services et clients.

Les services offerts par les objets sont, d'une part, des données, de type élémentaire ou structuré, que nous appellerons des *attributs*, et d'autre part, des actions que nous appellerons *méthodes*. Par exemple, un rectangle est un objet caractérisé par deux attributs, sa largeur et sa longueur, et des méthodes de calcul de sa surface ou de son périmètre.

Les langages de programmation par objets offrent des moyens de description des objets manipulés par le programme. Plutôt que de décrire individuellement chaque objet, ils fournissent une construction, la *classe*, qui décrit un ensemble d'objets possédant les mêmes propriétés. Une classe comportera en particulier la déclaration des données et des méthodes. Les langages de programmation à objets qui possèdent le concept de classe sont appelés *langages de classes*.

La déclaration d'une classe correspond à la déclaration d'un nouveau type. L'ensemble des rectangles pourra être décrit par une déclaration de classe comprenant deux attributs de type réel :

```
classe Rectangle
    largeur, longueur type réel
finclasse Rectangle
```

La classe `Rectangle` fournira comme service à ses clients, l'accès à ses attributs : sa longueur et sa largeur. Nous verrons dans la section 7.2 comment déclarer les méthodes. La déclaration d'une variable `r` de type `Rectangle` s'écrit de la façon habituelle :

```
variable r type Rectangle
```

7.1.1 Création des objets

Il est important de bien comprendre la différence entre les notions de classe et d'objet[1]. Pour nous, les classes sont des descriptions purement *statiques* d'ensembles possibles d'objets. Leur rôle est de définir de nouveaux types. En revanche, les objets sont des *instances* particulières d'une classe. Les classes sont un peu comme des « moules » de fabrication dont sont issus les objets. En cours d'exécution d'un programme, seuls les objets existent.

La déclaration d'une variable d'une classe donnée *ne crée pas* l'objet. L'objet, instance de la classe, doit être explicitement créé grâce à l'opérateur **`créer`**. *Chaque* objet créé possède tous les attributs de la classe dont il est issu. Chaque objet possède donc ses propres attributs, distincts des attributs d'un autre objet : deux rectangles ont chacun une largeur et une longueur qui leur est propre.

Les attributs de l'objet prennent des valeurs initiales données par un *constructeur*. Pour chaque classe, il existe un constructeur par défaut qui initialise les attributs à des valeurs initiales par défaut. Ainsi la déclaration suivante :

```
variable r type Rectangle créer Rectangle()
```

définit une variable `r` qui désigne un objet créé de type `Rectangle`, et dont les attributs sont initialisés à la valeur réelle 0 par le constructeur `Rectangle()`. La figure 7.1 montre une instance de la classe `Rectangle` avec ses deux attributs initialisés à 0. La variable `r` qui désigne l'objet créé est une *référence* à l'objet, et non pas l'objet lui-même.

Le fonctionnement d'une affectation d'objets de type classe n'est plus tout à fait le même que pour les objets élémentaires. Ainsi, l'affectation `q ←r`, affecte à `q` la référence à l'objet désigné par `r`. Après l'affectation, les références `r` et `q` désignent le même objet (voir la figure 7.2).

7.1.2 Destruction des objets

Que faire des objets qui ne sont plus utilisés ? Ces objets occupent inutilement de la place en mémoire, et il convient de la récupérer. Selon les langages de programmation, cette tâche

1. Dans la plupart des langages à objets, cette différence est plus subtile, puisqu'une classe peut être elle-même un objet.

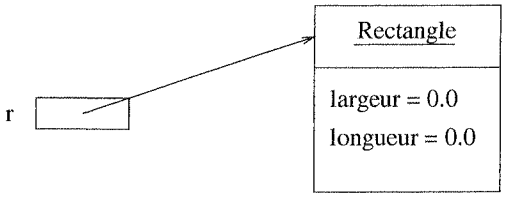


Figure 7.1 – *`r` désigne un objet de type* `Rectangle`.

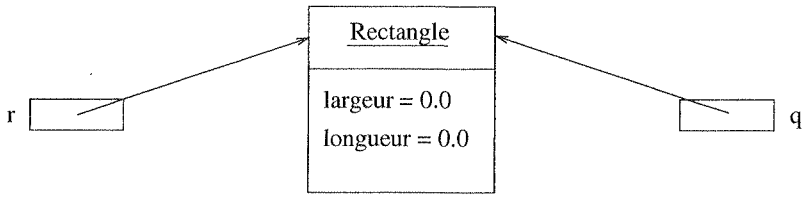


Figure 7.2 – *`r` et `q` désignent le même objet.*

est laissée au programmeur ou traitée automatiquement par le support d'exécution du langage. La première approche est dangereuse dans la mesure où elle laisse au programmeur la responsabilité d'une tâche complexe qui pose des problèmes de sécurité. Le programmeur est-il bien sûr que l'objet qu'il détruit n'est plus utilisé ? Au contraire, la seconde approche simplifiera l'écriture des programmes et offrira bien évidemment plus de sécurité.

Notre notation algorithmique ne tiendra pas compte de ces problèmes de gestion de la mémoire et aucune primitive ne sera définie.

7.1.3 Accès aux attributs

L'accès à un attribut d'un objet est valide si ce dernier existe, c'est-à-dire s'il a été créé au préalable. Cet accès se fait simplement en le *nommant*. Dans une classe, toute référence aux attributs définis dans la classe elle-même s'applique à l'instance courante de l'objet, que nous appellerons l'*objet courant*.

En revanche, l'accès aux attributs depuis des clients, *i.e.* d'autres objets, se fait par une notation pointée de la forme `n.a`, où `n` est un nom qui désigne l'objet, et `a` un attribut particulier.

La liste des services fournis aux clients est contrôlée explicitement par chaque objet. Par défaut, nous considérerons que tous les attributs d'une classe sont *privés*, et ne sont pas directement accessibles. Les attributs accessibles par les clients seront explicitement nommés dans une clause **`public`**. Pour rendre les deux attributs de la classe `Rectangle` publics, nous écrirons :

```
classe Rectangle
  public  largeur, longueur
  largeur, longueur type réel
finclasse Rectangle
```

Ainsi, dans cette classe, la notation `largeur` désigne l'attribut de même nom de l'objet courant. La notation `r.largeur` désigne l'attribut `largeur` de l'objet désigné par la variable `r` déclarée précédemment.

7.1.4 Attributs de classe partagés

Nous avons vu que chaque objet possède ses propres attributs. Toutefois, il peut être intéressant de partager un attribut de classe par toutes les instances d'une classe. Nous appellerons attribut *partagé*, un attribut qui possède cette propriété. Un tel attribut possédera une information représentative de classe tout entière. Le mot-clé **`partagé`** devra explicitement précéder la déclaration des attributs partagés.

7.1.5 Les classes en Java

La syntaxe des classes en `uit celle` de classes données ci-dessus dans notre pseudo-langage. La classe `Rectangle` s'écrira :

```
class Rectangle {
  public double largeur, longueur;
}
```

La création d'un objet se fait grâce à l'opérateur **`new`** et son initialisation grâce à un constructeur dont le nom est celui de la classe. La variable `r` précédente est déclarée et initialisée en JAVA de la façon suivante :

```
Rectangle r = new Rectangle();
```

La gestion de la destruction des objets est laissée à la charge de l'interprète du langage JAVA. Les objets qui deviennent inutiles, parce qu'ils ne sont plus utilisés, sont automatiquement détruits par le support d'exécution. Toutefois, il est possible de définir dans chaque classe la méthode `finalize`, appelée immédiatement avant la destruction de l'objet, afin d'exécuter des actions d'achèvement particulières.

L'accès aux attributs d'un objet, appelés *membres* dans la terminologie JAVA, se fait par la notation pointée vue précédemment. L'autorisation d'accès à un membre par des clients est explicitée grâce au mot-clé **`public`** ou **`private`**[2] placé devant sa déclaration.

Par convention, l'objet courant est désigné par le nom **`this`**. Ainsi, les deux notations `largeur` et **`this`**`.largeur` désignent le membre `largeur` de l'objet courant. On peut considérer que chaque objet possède un attribut **`this`** qui est une référence sur lui-même.

Un membre précédé du mot-clé **`static`** le rend partagé par tous les objets de la classe. On peut accéder à un membre **`static`** même si *aucun* objet n'a été créé, en utilisant dans la notation pointée le nom de classe. Ainsi, dans l'instruction `System.out.println()`, `System` est le nom d'une classe dans laquelle est déclarée la variable statique `out`.

2. En fait, JAVA définit deux autres types d'accès aux membres d'une classe, mais nous n'en parlerons pas pour l'instant.

7.2 LES MÉTHODES

Le second type de service offert par un objet à ses clients est un ensemble d'opérations sur les attributs. Le rôle de ces opérations, appelées *méthodes*, est de donner le modèle opérationnel de l'objet.

Nous distinguerons deux types de méthodes, les procédures et les fonctions. Leurs déclarations et les règles de transmission des paramètres suivent les mêmes règles que celles énoncées au chapitre 6. Toutefois, nous considérerons, d'une part, que les procédures réalisent une action sur l'état de l'objet, en modifiant les valeurs des attributs de l'objet, et d'autre part, que les fonctions se limitent à renvoyer un état de l'objet. Les procédures modifieront directement les attributs de l'objet sans utiliser de paramètres résultats.

Notez que ce modèle ne pourra pas toujours être suivi à la lettre. Certaines fonctions pourront être amenées à modifier des attributs, et des procédures à ne pas modifier l'état de l'objet.

Pour toutes les instances d'une même classe, il n'y a qu'un exemplaire de chaque méthode de la classe. Contrairement aux attributs d'instance qui sont propres à chaque objet, les objets d'une classe partagent la même méthode d'instance.

Nous pouvons maintenant compléter notre classe `Rectangle` avec, par exemple, deux fonctions qui renvoient le périmètre et la surface du rectangle, et deux procédures qui modifient respectivement la largeur et la longueur du rectangle courant. Notez que la définition de ces deux dernières méthodes n'est utile que si les attributs sont privés.

```
classe Rectangle
  public périmètre, surface, changerLargeur, changerLongueur
  {les attributs}
  largeur, longueur type réel

  {les méthodes}
  fonction périmètre() : réel
  {Rôle: retourne le périmètre du rectangle}
      rendre 2×(largeur+longueur)
  finfonc {périmètre}

  fonction surface() : réel
  {Rôle: retourne la surface du rectangle}
      rendre largeur×longueur
  finfonc {surface}

  procédure changerLargeur(donnée lg : réel)
  {Rôle: met la largeur du rectangle à lg}
      largeur ← lg
  finproc

  procédure changerLongueur(donnée lg : réel)
  {Rôle: met la longueur du rectangle à lg}
      longueur ← lg
  finproc
finclasse Rectangle
```

Si les méthodes possèdent des en-têtes différents, bien qu'elles aient le *même* nom, elles sont considérées comme distinctes et dites *surchargées*. La notion de surcharge est normalement utilisée lorsqu'il s'agit de définir plusieurs mises en œuvre d'une même opération. La plupart des langages de programmation la proposent implicitement pour certains opérateurs ; traditionnellement, l'opérateur + désigne l'addition entière et l'addition réelle, ou encore la concaténation de chaînes de caractères, comme en JAVA. En revanche, peu de langages proposent aux programmeurs la surcharge des fonctions ou des procédures. Le langage EIFFEL l'interdit même, arguant que donner aux programmeurs la possibilité du choix d'un même nom pour deux opérations différentes est une source de confusion. Dans une classe, chaque propriété doit posséder un nom unique[3].

7.2.1 Accès aux méthodes

L'accès aux méthodes suit les mêmes règles que celles aux attributs. Dans la classe, toute référence à une méthode s'applique à l'instance courante de l'objet, et depuis un client il faudra utiliser la notation pointée. Pour rendre les méthodes accessibles aux clients, il faudra, comme pour les attributs, les désigner publiques.

```
r.changerLargeur(2.5)
r.changerLongueur(7)
{r désigne un rectangle de largeur 2.5 et de longueur 7}
```

7.2.2 Constructeurs

Nous avons déjà vu que lors de la création d'un objet, les attributs étaient initialisés à des valeurs par défaut, grâce à un constructeur par défaut. Mais, il peut être utile et nécessaire pour une classe de proposer son propre constructeur. Il est ainsi possible de redéfinir le constructeur par défaut de la classe `Rectangle` pour initialiser les attributs à des valeurs autres que zéro.

```
classe Rectangle
  public
        périmètre, surface, changerLargeur, changerLongueur
  {les attributs}
  largeur, longueur type réel
  constructeur Rectangle()
     largeur ← 1
     longueur ← 1
  fincons
  {les méthodes}
   ...
finclasse Rectangle
```

Bien souvent, il est souhaitable de proposer plusieurs constructeurs aux utilisateurs d'une classe. Malgré les remarques précédentes, nous considérerons que toutes les classes peuvent définir un ou plusieurs constructeurs, dont les noms sont celui de la classe, selon le mécanisme de surcharge. La classe `Rectangle` pourra définir, par exemple, un second constructeur

3. Lire avec intérêt les arguments de B. MEYER [Mey97] contre le mécanisme de surcharge.

pour permettre aux clients de choisir leurs propres valeurs initiales. Les paramètres d'un constructeur sont implicitement des paramètres « données ».

```
constructeur Rectangle(données lar, lon : réel)
    largeur ← lar
    longueur ← lon
fincons
```

Les créations de rectangles suivantes utilisent les deux constructeurs définis précédemment :

```
variables
        r type Rectangle créer Rectangle()
        {r désigne un rectangle (1,1)}
        s type Rectangle créer Rectangle(3,2)
        {s désigne un rectangle (3,2)}
```

7.2.3 Les méthodes en Java

La déclaration de la classe `Rectangle` s'écrit en JAVA comme suit :

```
class Rectangle {
    public double largeur, longueur;
    // les constructeurs
    public Rectangle() {
        largeur = longueur = 1;
    }
    public Rectangle(double lar, double lon) {
        largeur = lar;  longueur = lon;
    }
    // les méthodes
    public double périmètre() {
    // Rôle: retourne le périmètre du rectangle
      return 2*(largeur+longueur);
    }
    public double surface() {
    // Rôle: retourne la surface du rectangle
      return largeur*longueur;
    }
    public void changerLargeur(double lg) {
    // Rôle: met la largeur du rectangle à lg
      largeur = lg;
    }
    public void changerLongueur(double lg) {
    // Rôle: met la longueur du rectangle à lg
      longueur = lg;
    }
} // fin classe Rectangle
```

Les déclarations des méthodes débutent par le type du résultat renvoyé pour les fonctions, ou par **void** pour les procédures. Suit le nom de la méthode et ses paramètres formels pla-

cés entre parenthèses. Les noms des paramètres sont précédés de leur type, comme pour les attributs, et séparés par des virgules.

La transmission des paramètres se fait *toujours par valeur*. Mais, précisons que dans le cas d'un objet non élémentaire (*i.e.* défini par une classe), la valeur transmise est la *référence* à l'objet, et *non pas* l'objet lui-même.

La méthode `changerLongueur` est une procédure à un paramètre transmis par valeur, et la méthode `périmètre` est une fonction sans paramètre qui retourne un réel double précision.

Le corps d'une méthode est parenthésé par des accolades ouvrantes et fermantes. Il contient une suite de déclarations locales et d'instructions. Dans une fonction, l'instruction **`return`** `e;` termine l'exécution de la fonction, et retourne le résultat de l'évaluation de l'expression `e`. Le type de l'expression `e` doit être compatible avec le type de valeur de retour déclaré dans l'en-tête de la fonction. Une procédure peut également exécuter l'instruction **`return`**, mais sans évaluer d'expression. Son effet sera simplement de terminer l'exécution de la procédure.

Les déclarations des constructeurs suivent les règles précédentes, mais le nom du constructeur n'est précédé d'aucun type, puisque c'est lui-même un nom de type (*i.e.* celui de la classe). Notez que le constructeur de l'objet courant est désigné par **`this`**`()`.

Les méthodes et les constructeurs peuvent être surchargés dans une même classe. Leur distinction est faite sur le nombre et le type des paramètres, mais, attention, pas sur celui du type du résultat.

JAVA permet de déclarer une méthode statique en faisant précéder sa déclaration par le mot-clé **`static`**. Comme pour les attributs, les méthodes statiques existent indépendamment de la création des objets, et sont accessibles en utilisant dans la notation pointée le nom de classe. La méthode `main` est un exemple de méthode statique. Elle doit être déclarée statique puisqu'aucun objet de la classe qui la contient n'est créé. Notez que la notion de méthode statique remet en cause le modèle objet, puisqu'il est alors possible d'écrire un programme JAVA sans création d'objet, et exclusivement structuré autour des actions.

7.3 ASSERTIONS SUR LES CLASSES

Nous avons déjà dit que la validité des programmes se démontre de façon formelle, à l'aide d'assertions. Les assertions sur les actions sont des affirmations sur l'état du programme avant et après leur exécution.

De même, il faudra donner des assertions pour décrire les propriétés des objets. B. MEYER[4] nomme ces assertions des *invariants de classe*. Un invariant de classe est un ensemble d'affirmations, mettant en jeu les attributs et les méthodes publiques de la classe, qui décrit les propriétés de l'objet. L'invariant de classe doit être vérifié :

- après l'appel d'un constructeur ;
- avant et après l'exécution d'une méthode publique.

4. *ibidem.*

Dans la mesure où les dimensions de rectangle doivent être positives, un invariant possible pour la classe `Rectangle` est :

{largeur ⩾ 0 et longueur ⩾ 0}

Le modèle de programmation contractuelle établit une relation entre une classe et ses clients, qui exprime les droits et les devoirs de chaque partie. Ainsi, une classe peut dire à ses clients :

- Mon invariant de classe est vérifié. Si vous me promettez de respecter l'antécédent de la méthode *m* que vous désirez appeler, je promets de fournir un état final conforme à l'invariant de classe *et* au conséquent de *m*.
- Si vous respectez l'antécédent du constructeur, je promets de créer un objet qui satisfait l'invariant de classe.

Si ce contrat passé entre les classes et les clients est respecté, nous pourrons garantir la validité du programme, c'est-à-dire qu'il est conforme à ses spécifications. Toutefois, deux questions se posent. Comment vérifier que le contrat est effectivement respecté ? Et que se passe-t-il si le contrat n'est pas respecté ?

La vérification doit se faire de façon automatique, le langage de programmation devant offrir des mécanismes pour exprimer les assertions et les vérifier. Il est à noter que très peu de langages de programmation offrent cette possibilité.

Si le contrat n'est pas respecté[5], l'exécution du programme provoquera une erreur. Toutefois, peut-on quand même poursuivre l'exécution du programme lorsque la rupture du contrat est avérée, tout en garantissant la validité du programme ? Nous verrons au chapitre 13, comment la notion d'exception apporte une solution à ce problème.

7.4 EXEMPLES

Dans le chapitre précédent, nous avons vu comment structurer en sous-programmes la résolution d'une équation du second degré et le calcul de la date du lendemain. Nous reprenons ces deux exemples en les réorganisant autour des objets.

7.4.1 Équation du second degré

L'objet central de ce problème est l'équation. On considère que chaque équation porte ses racines. Ainsi, chaque objet de type `Éq2Degré` possède comme attributs les solutions de l'équation, qui sont calculées par le constructeur. Ce constructeur possède trois paramètres qui sont les trois coefficients de l'équation. La classe fournit une méthode pour afficher les solutions.

```
classe Éq2Degré
{Invariant de classe:  ax²+bx+c=0 avec a≠0}
public  affichersol
```

5. Si l'on suppose que les classes sont justes, ce sont les antécédents des méthodes appelées qui ne seront pas respectés.

```
r1, r2, i1, i2 type réel

constructeur Éq2Degré(données a, b, c : réel)
{Antécédent :  a≠0}
{Conséquent : (x - (r1+i×i1)) (x - (r2 +i×i2)) = 0}
     résoudre(a,b,c)
fincons

procédure résoudre(données a, b, c : réel)
{Antécédent: a, b, c coefficients réels de l'équation
             ax²+bx+c=0 et a≠0}
{Conséquent: (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
 constante
       ε = ? {dépend de la précision des réels sur la machine}
 variable Δ type réel {le discriminant}

     Δ ←carré(b)-4×a×c
     si Δ⩽0 alors {calcul des racines réelles}
        si b>0 alors r1 ← -(b+√Δ)/(2×a)
           sinon r1 ← (√Δ-b)/(2×a)
        finsi
        {r1 est la racine la plus grande en valeur absolue}
        si |r1|< ε alors r2 ←0
                   sinon r2 ← c/(a×r1)
        finsi
        i1 ← 0 i2 ← 0
        {(x-r1)(x-r2)=0}
     sinon  {calcul des racines complexes}
        r1 ← r2 ← -b/(2×a)
        i1 ← √-Δ/(2×a)
        i2 ← -i1
     finsi
     {(x-(r1+ii1)) (x-(r2+ii2)) = 0}
 finproc {résoudre}

 procédure affichersol()
  {Antécédent:  (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
  {Conséquent : les solutions (r1,i1) et (r2,i2) sont
                affichées sur la sortie standard}
      écrire(r1,i1)
      écrire(r2,i2)
  finproc
finclasse Éq2Degré
```

La traduction en JAVA de cet algorithme est immédiate et ne pose aucune difficulté :

```
class Éq2Degré {
    // Invariant de classe: ax²+bx+c=0 avec a≠0
    private double r1, r2, i1, i2;
```

```
    public Éq2Degré(double a, double b, double c)
    // Antécédent: a≠0
    // Conséquent: (x - (r1+i×i1)) (x - (r2+i×i2)) = 0
    { résoudre(a, b, c); }

    public void résoudre(double a, double b, double c)
    // Antécédent: a, b, c coefficients réels de l'équation
    // ax²+bx+c=0 et avec a≠0
    // Conséquent: (x-(r1+i×i1)) (x-(r2+i×i2)) = 0
    {
        final double ε = 1E-100;
        final double Δ = (b*b)-4*a*c;

        if (Δ>=0) { // calcul des racines réelles
            if (b>0) r1 = -(b+Math.sqrt(Δ))/(2*a);
            else r1 = (Math.sqrt(Δ)-b)/(2*a);
            // r1 est la racine la plus grande en valeur absolue
            r2 = Math.abs(r1) < ε ? 0 : c/(a*r1);
            i1=i2=0;
            // (x-r1)(x-r2)=0
        }
        else { // calcul des racines complexes
            r1 = r2 = -b/(2*a);
            i1=Math.sqrt(-Δ)/(2*a); i2=-i1;
        }
        // (x-(r1+ii1)) (x-(r2+ii2)) = 0
    }
    public String toString()
    // Conséquent: renvoie une chaîne de caractères
    // qui contient les deux racines
    {
        return "r1=␣(" + r1 + "," + i1 + ")\n" +
               "r2=␣(" + r2 + "," + i2 + ")";
    }
} // fin classe Éq2Degré
```

Vous notez la présence de la méthode `toString` qui permet la conversion d'un objet `Éq2Degré` en une chaîne de caractères[6]. Celle-ci peut être utilisée implicitement par certaines méthodes, comme par exemple, la procédure `System.out.println` qui n'accepte en paramètre qu'une chaîne de caractères. Si le paramètre n'est pas de type `String`, deux cas se présentent : soit il est d'un type de base, et alors il est converti implicitement ; soit le paramètre est un objet, et alors la méthode `toString` de l'objet est appelée afin d'obtenir une représentation sous forme de chaîne de caractères de l'objet courant.

Nous pouvons écrire la classe `Test`, contenant la méthode `main`, pour tester la classe `Éq2Degré`.

6. Une chaîne de caractères est un objet constitué par une suite de caractères. Cette notion est présentée à la section 9.7 page 93.

```
import java.io.*;
class Test {
    public static void main (String[] args) throws IOException
    {
        Éq2Degré e = new Éq2Degré(StdInput.readDouble(),
                                  StdInput.readDouble(),
                                  StdInput.readlnDouble());
        // écrire les racines solutions sur la sortie standard
        System.out.println(e);
    }
} // fin classe Test
```

La variable e désigne un objet de type Éq2Degré. Les trois paramètres de l'équation sont lus sur l'entrée standard et passés au constructeur à la création de l'objet. L'appel de la méthode println a pour effet d'écrire sur la sortie standard les deux racines de l'équation e.

7.4.2 Date du lendemain

L'objet central du problème est bien évidemment la date. Nous définissons une classe Date, dont les principaux attributs sont trois entiers qui correspondent au jour, au mois et à l'année. Cette classe offre à ses clients les services de calcul de la date du lendemain et d'affichage (en toutes lettres) de la date. Notez que la liste de services offerte par cette classe n'est pas figée ; si nécessaire, elle pourra être complétée ultérieurement.

Le calcul de la date du lendemain modifie l'objet courant en ajoutant un jour à la date courante. La vérification de la validité de la date sera faite par le constructeur au moment de la création de l'objet. Ainsi, l'invariant de classe affirme que la date courante, représentée par les trois attributs jour, mois et année, constitue une date valide supérieure ou égale à une année minima. L'attribut qui définit cette année minima est une constante publique.

```
classe Date
 {Invariant de classe: les attributs jours, mois et année
                       représentent une date valide ⩾ annéeMin}
 public demain, afficher, annéeMin

 jour, mois, année type entier
 constante annéeMin = 1582

 constructeur Date(données j, m, a : entier)
 {Antécédent:}
 {Conséquent: jour = j, mois = m et année = a
              représentent une date valide}
 ...
 fincons

 procédure demain()
 {Antécédent : jour, mois, année représentent une date valide}
 {Conséquent : jour, mois, année représentent la date du lendemain}
 ...
 finproc
```

```
 procédure afficher()
 {Conséquent : la date courante est affichée sur la sortie standard}
  ...
 finproc
finclasse Date
```

Nous n'allons pas récrire les corps du constructeur et des deux méthodes dans la mesure où leurs algorithmes donnés page 57 restent les mêmes. Toutefois, le calcul de la date du lendemain nécessite de connaître le nombre de jours maximum dans le mois et de vérifier si l'année est bissextile. Le nombre de jours maximum, `nbJoursMois`, sera un attribut déclaré privé. Enfin, la fonction `bissextile` sera une méthode que l'on pourra définir publique, puisque d'un usage général. Voici, l'écriture en JAVA de la classe `Date`.

```
class Date {
  // Invariant de classe: les attributs jour, mois et année
  // représentent une date valide ⩾ annéeMin
  private static final int AnnéeMin = 1582;
  private int jour, mois, année;
  private int nbJoursMois;
  public Date(int j, int m, int  a)
  // Conséquent: jour = j, mois = m et année = a
  // représentent une date valide
  {
     if (a < AnnéeMin) {
         System.err.println("Année incorrecte");
         System.exit(1);
     }
     // l'année est bonne
     année=a;
     if (m<1 || m>12) {
          System.err.println("Mois incorrect");
          System.exit(1);
     }
     // le mois est bon, tester le jour
     mois=m;
     switch (m) {
       case 1 :
       case 3 :
       case 5 :
       case 7 :
       case 8 :
       case 10 :
       case 12 : nbJoursMois=31; break;
       case 4 :
       case 6 :
       case 9 :
       case 11 : nbJoursMois=30; break;
       case 2 : nbJoursMois = bissextile() ? 29 : 28;
     }
```

```
    if (j>nbJoursMois) {
        System.err.println("Jour incorrect");
        System.exit(1);
    }
    // le jour j est valide
    jour=j;
}
public void demain()
// Antécédent: jour, mois, année représentent une date valide
// Conséquent: jour, mois, année représentent la date du lendemain
{
    // jour ⩽ NbreJours et mois dans [1,12]
    if (jour<nbJoursMois)
        // le mois et l'année ne changent pas
        jour++;
    else {
        // c'est le dernier jour du mois, il faut passer au
        // premier jour du mois suivant
        jour=1;
        if (mois<12) mois++;
        else {
            // c'est le dernier mois de l'année, il faut passer
            // au premier mois de l'année suivante
            mois=1; année++;
        }
     }
}
public boolean bissextile()
// Rôle: teste si l'année courante est bissextile ou non
{
    return  année%4 == 0 && année%100 != 0
            || année%400 == 0;
}
public String toString()
// Conséquent: renvoie la date courante sous forme
// d'une chaîne de caractères
{
    String smois="";

    switch (mois) {
      case 1  : smois = "janvier";   break;
      case 2  : smois = "février";   break;
      case 3  : smois = "mars";      break;
      case 4  : smois = "avril";     break;
      case 5  : smois = "mai";       break;
      case 6  : smois = "juin";      break;
      case 7  : smois = "juillet";   break;
      case 8  : smois = "aout";      break;
      case 9  : smois = "septembre"; break;
      case 10 : smois = "octobre";   break;
```

```
        case 11 : smois = "novembre";  break;
        case 12 : smois = "décembre";
      }
      return jour + "␣" + smois + "␣" + année;
    }
} // fin classe Date
```

Le constructeur `Date` vérifie la validité de la date. Si la date est incorrecte, il se contente d'écrire un message sur la sortie d'erreur et d'arrêter brutalement le programme.

La classe de test donnée ci-dessous crée un objet de type `Date`. Le jour, le mois et l'année sont lus sur l'entrée standard. Elle calcule la date du lendemain et l'affiche.

```
import java.io.*;
class DateduLendemain {
  public static void main (String[] args) throws IOException
  {
      Date d = new Date(StdInput.readInt(),
                        StdInput.readInt(),
                        StdInput.readlnInt());
      d.demain();
      System.out.println(d);
  }
} // fin classe DateduLendemain
```

7.5 EXERCICES

Exercice 7.1. Ajoutez à la classe `Éq2Degré` les fonctions `premièreRacine` et `secondeRacine` qui retournent, respectivement, la première et la seconde racine de l'équation.

Exercice 7.2. Définissez une classe `Complexe`, pour représenter les nombres de l'ensemble $\mathbb{C}$. Un objet complexe aura deux attributs, une partie réelle et une partie imaginaire. Vous définirez un constructeur par défaut qui initialisera les deux attributs à zéro, ainsi qu'un constructeur qui initialisera un nombre complexe à partir deux paramètres réels. Vous écrirez la méthode `toString` qui donne une représentation d'un nombre complexe sous la forme `(r,i)`.

Exercice 7.3. Utilisez votre classe `Complexe` pour représenter les racines solutions de l'équation du second degré.

Exercice 7.4. Complétez la classe `Complexe` avec les opérations d'addition et de multiplication. Notez que la multiplication est plus simple à écrire si les complexes sont représentés sous forme polaire. On rappelle que tout complexe z admet une représentation cartésienne $x + iy$ et polaire $\rho e^{i\theta}$ où ρ est le module et θ l'argument de z. L'argument n'est défini que si $z \neq 0$. Le passage d'un système de coordonnées à l'autre se fait à l'aide des formules de conversion :

coordonnées polaires	coordonnées cartésiennes
$\rho = \sqrt{x^2 + y^2}$	$x = \rho\cos(\theta)$
$\theta = atan(y/x)$	$y = \rho\sin(\theta)$

Le produit de deux complexes $z1$ et $z2$ est donné par les deux équations suivantes :

$$\begin{aligned} \rho(z1 \times z2) &= \rho(z1) \times \rho(z2) \\ \theta(z1 \times z2) &= \theta(z1) + \theta(z2) \end{aligned}$$

Exercice 7.5. Complétez la classe `Date` avec les méthodes suivantes :

- `hier` qui soustrait un jour à la date courante ;
- `avant`, `après`, `égale` qui teste si une date passée en paramètre est, respectivement, antérieure, postérieure ou égale à la date courante.
- `joursÉcoulés` qui retourne le nombre de jours écoulés entre la date courante et une date passée en paramètre.
- `nomDuJour` qui retourne le nom du jour de la semaine de la date courante.

Chapitre 8

Énoncés itératifs

Jusqu'à présent, nous avons vu qu'un énoncé ne pouvait être exécuté que zéro ou une fois. Est-il possible de répéter plus d'une fois l'exécution d'une énoncé ? La réponse est oui, grâce aux énoncés *itératifs*.

8.1 FORME GÉNÉRALE

Un énoncé itératif, comme le montre sa représentation circulaire donnée par la figure 8.1, est souvent appelé *boucle*. On entre dans la boucle avec l'antécédent P_a, et à chaque tour de boucle les énoncés qui la composent sont exécutés. L'arrêt de la boucle se produira lorsque le prédicat d'achèvement B sera vérifié.

L'affirmation P_a est l'antécédent de la boucle, P_c le conséquent, et P_0, P_1, ..., P_n des affirmations toujours vraies quel que soit le nombre d'itérations. Ces dernières sont appelées *invariants* de boucle. La règle de déduction de l'énoncé itératif s'exprime comme suit :

si $\{P_0\} \overset{A_1}{\Rightarrow} \{P_1\} \overset{A_2}{\Rightarrow} \{P_2\}$ ... $\{P_{i-1}\} \overset{A_i}{\Rightarrow} \{P_i\} \overset{A_{i+1}}{\Rightarrow} \{P_{i+1}\}$ ... $\{P_{n-1}\} \overset{A_n}{\Rightarrow} \{P_n\}$
et $\{P_n\} \Rightarrow \{P_0\}$ et $\{P_a\} \Rightarrow \{P_0\}$
et $\{P_i$ *et* $B\} \Rightarrow \{P_c\}$ et $\{P_i$ *et non* $B\} \Rightarrow \{A_{i+1}\}$
alors $\{P_a\}$ énoncé-itératif-général $\{P_c\}$

L'emplacement spécifique de l'affirmation P_i, *i.e.* juste avant le prédicat d'achèvement, lui confère un rôle privilégié. On considérera cette affirmation comme l'*invariant* de boucle. Cet invariant décrit la sémantique de l'énoncé itératif, et doit être un préalable, ce qui n'est pas toujours évident, à la programmation de la boucle. Une syntaxe possible d'un énoncé itératif général est la suivante :

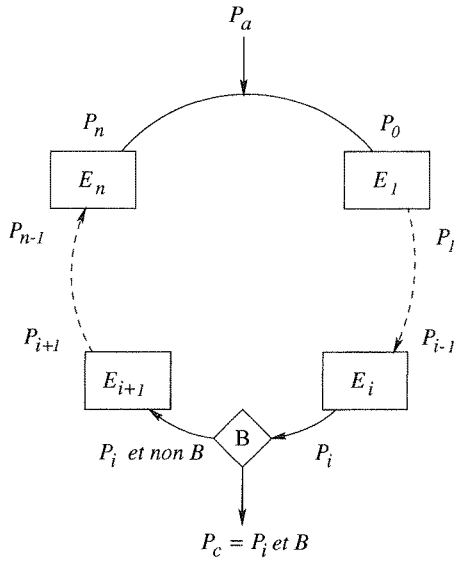


Figure 8.1 – *Forme générale d'un énoncé itératif.*

```
itérer
      E1
      arrêt si B
      E2
finitérer
```

Sa règle de déduction est donnée ci-dessous. L'invariant de boucle est l'affirmation Q.

$\underline{\texttt{si}}\ \{P\} \overset{E_1}{\Rightarrow} \{Q\}\ \underline{\texttt{et}}\ \{Q\ et\ non\ B\} \overset{E_2}{\Rightarrow} \{P\}$
$\underline{\texttt{alors}}\ \{P\}$ **itérer** E1 **arrêt si** B E2 **finitérer** $\{Q\ et\ B\}$

À partir de la forme générale de l'énoncé itératif, deux énoncés particuliers peuvent être déduits selon que l'ensemble des énoncés de A_1 à A_i ou l'ensemble des énoncés de A_{i+1} à A_n est vide. Ces deux énoncés itératifs, qui existent dans la plupart des langages de programmation, sont les énoncés **tantque** et **répéter**.

8.2 L'ÉNONCÉ TANTQUE

Lorsque les énoncés A_1 à A_i sont vides, nous sommes en présence d'un énoncé **tantque**. Sa syntaxe est généralement :

```
tantque B faire E fintantque
```

où le prédicat d'achèvement B est une expression booléenne. Tant que l'évaluation de l'expression B retourne la valeur *vrai*, l'énoncé E est exécuté. La boucle s'achève lorsque B est

faux. Si le prédicat d'achèvement est immédiatement vérifié, l'énoncé E n'est pas exécuté. L'énoncé E peut donc être exécuté zéro ou plusieurs fois. Plus formellement, la règle de déduction de cet énoncé **tantque** est :

si $\{P \text{ et } B\} \overset{E}{\Rightarrow} \{P\}$
alors $\{P\}$ **tantque** B **faire** E **fintantque** $\{P \text{ et non } B\}$

L'énoncé tantque de JAVA

Cet énoncé s'écrit selon une syntaxe classique. Notez cependant, la présence de parenthèses autour du prédicat booléen B, et l'absence de mot-clé avant l'énoncé E.

```
while (B) E
```

8.3 L'ÉNONCÉ RÉPÉTER

Cette fois-ci, les énoncés A_{i+1} à A_n sont vides. L'énoncé itératif s'appelle l'énoncé **répéter** et sa syntaxe est la suivante :

répéter E **jusqu'à** B

Le prédicat d'achèvement B est là encore une expression booléenne. L'énoncé E est exécuté jusqu'à ce que l'évaluation de l'expression B retourne la valeur *vrai*. Tant que la valeur est *faux*, l'énoncé E est exécuté. Notez que l'énoncé E est exécuté *au moins* une fois. Ainsi, le choix entre l'utilisation d'un énoncé **tantque** et **répéter** dépendra du nombre d'itérations minimum à effectuer. Plus formellement, la règle de déduction de l'énoncé **répéter** est donnée ci-dessous. Notez que l'invariant de boucle est l'affirmation Q et non pas P !

si $\{P\}$ E $\{Q\}$ et $\{Q \text{ et non } B\}$ E $\{Q\}$
alors $\{P\}$ **répéter** E **jusqu'à** B $\{Q \text{ et } B\}$

Un langage de programmation pourrait se limiter à la seule définition de l'énoncé itératif **tantque**. En effet, tous les autres énoncés itératifs peuvent s'exprimer à partir de celui-ci. Par exemple, les énoncés **itérer** et **répéter** s'écrivent comme suit :

itérer $\Leftrightarrow$ E_1 **tantque non** B **faire** E_2 E_1 **fintantque**
répéter $\Leftrightarrow$ E **tantque non** B **faire** E **fintantque**

L'énoncé répéter de JAVA

Cet énoncé s'écrit de la façon suivante :

```
do E while (B);
```

Notez que la sémantique du prédicat d'achèvement est à l'opposé de celle de la forme algorithmique. L'itération s'achève lorsque le prédicat B prend la valeur *faux*. La règle de déduction est donc légèrement modifiée :

si $\{P\}$ E $\{Q\}$ et $\{Q \text{ et } B\}$ E $\{Q\}$
alors $\{P\}$ **do** E **while** (B); $\{Q \text{ et non } B\}$

8.4 FINITUDE

Lorsqu'un programme exécute une boucle, il faut être certain que le processus itératif s'achève ; sinon on dit que le programme *boucle*. La *finitude* des énoncés itératifs est une propriété fondamentale des programmes informatiques. Une façon de démontrer la finitude d'une boucle est de rechercher une fonction $f(X)$ positive qui décroît strictement vers une valeur particulière. Pour cette valeur, par exemple zéro, on en déduit que B est vérifié, et que la boucle s'achève. X représente des variables mises en jeu dans le corps de la boucle, et qui devront nécessairement être modifiées par le processus itératif.

8.5 EXEMPLES

La première partie de ce chapitre était assez théorique. Il est temps de montrer l'utilisation des énoncés répétitifs sur des exemples concrets.

8.5.1 Factorielle

Nous désirons écrire la fonction `factorielle` qui calcule la factorielle d'un entier naturel passé en paramètre. Rappelons que la factorielle d'un entier naturel n est égale à :

$$n! = 1 \times 2 \times 3 \times \cdots \times (i-1) \times i \times \ldots n$$

Une première façon de procéder est de calculer une suite croissante de produits de 1 à n. Peut-on déterminer immédiatement l'invariant de boucle ? Au i^{e} produit, c'est-à-dire à la i^{e} itération qu'a-t-on calculé ? Réponse $i!$ et ce sera notre invariant. L'algorithme et la démonstration de sa validité sont donnés ci-dessous :

```
fonction factorielle(donnée n: naturel): naturel
{Antécédent : n ⩾ 0}
{Conséquent : factorielle = n!}
variables
   i, fact type naturel

   i ← 0
   fact ← 1 {Invariant : fact = i!}
   tantque i<n faire
       {fact × (i+1) = i! × (i+1) et i<n}
       i ← i+1
       {fact × i = i!}
       fact ← fact×i
       {fact = i!}
   fintantque
   {i = n et fact = i! = n!}
   rendre fact
finfonc {factorielle}
```

Une fonction qui permet de démontrer la finitude de la boucle est $f(i) = n - i$. Lorsque $i = n$, le prédicat d'achèvement est vérifié.

8.5.2 Minimum et maximum

On désire trouver le minimum et le maximum d'une suite (non vide) d'entiers $x_1, x_2, x_3, \ldots x_n$. Un entier de la suite est lu à chaque itération, et le calcul du minimum et du maximum se fait progressivement. À la i^e itération, on affirme que $\forall k \in [1,i]$, min $\leqslant x_k$ et max $\geqslant x_k$. Puisqu'il y a au moins un entier à lire sur l'entrée standard, nous utiliserons un énoncé **répéter**. Les valeurs initiales de min et max doivent être telles que tout x_1 est inférieur à min et supérieur à max.

```
procédure MinMax(donnée n : naturel
                 résultats min, max: entier)
{Antécédent : n > 0}
{Conséquent : min et max respectivement minimum et maximum
              d'une suite d'entiers lue sur l'entrée standard}
variable i type [0,n]

    min ← +∞
    max ← −∞
    i ← 0
    répéter
          i ← i+1
          lire(x)
          si x < min alors min ← x finsi
          {∀k ∈ [1,i], min⩽xk}
          si x > max alors max ← x finsi
          {∀k ∈ [1,i], min⩽xk et max⩾xk}
    jusqu'à i = n
    {∀k ∈ [1,i], min⩽xk et max⩾xk et i=n}
finproc {MinMax}
```

Comme pour factorielle, la fonction $f(i) = n - i$ permet de démontrer la finitude de la boucle.

8.5.3 Division entière

Rappelons tout d'abord, la définition de la division entière de deux entiers naturels a et b :

$$\forall a \geqslant 0, b > 0, a = \text{quotient} \times b + \text{reste}\ , 0 \leqslant r < b$$

Pour calculer la division entière, nous allons procéder par soustractions successives de la valeur b à la valeur a jusqu'à ce que le résultat de la soustraction soit inférieur à b. D'après ce que nous venons de dire l'invariant est la définition même de la division entière.

```
procédure DivisionEntière(données a, b : naturel
                          résultats quotient, reste : naturel)
{Antécédent : a⩾0, b>0}
{Conséquent : a = quotient×b + reste, 0⩽reste<b}
   reste ← a
   quotient ← 0 {Invariant : a = quotient × b + reste}
```

```
    tantque reste ⩾ b faire
        {a = (quotient+1) × b + reste - b et reste>b}
        quotient ← quotient+1
        {a = quotient × b + reste - b et reste⩾b}
        reste ← reste-b
        {a = quotient × b + reste}
    fintantque
    {a = quotient × b + reste et 0⩽reste<b}
finproc {DivisionEntière}
```

La fonction $f(reste) = reste - b$ garantit la finitude de la boucle.

8.5.4 Plus grand commun diviseur

Le plus grand commun diviseur de deux nombres naturels est l'entier naturel le plus grand qui les divise tous les deux. Il est tel que :

$$pgcd(a,b) = pgcd(a-b,b) \text{ et } a > b = pgcd(a,b-a) \text{ et } a < b$$

L'algorithme proposé par EUCLIDE [1] procède par soustractions successives des valeurs a et b, jusqu'à ce que a et b soient égaux.

```
fonction pgcd(données a, b: naturel): naturel
{Antécédent : a>0 et b>0}
{Conséquent : pgcd(a,b) = pgcd(a-b,b) et a>b
                        = pgcd(a,b-a) et a<b}

    tantque a≠b faire
        {pgcd(a,b) = pgcd(a-b,b) et a>b
                   = pgcd(a,b-a) et a<b}
        si a>b alors
            {pgcd(a,b) = pgcd(a-b,b) et a>b}
            a ← a-b
        sinon
            b ← b-a
            {pgcd(a,b) = pgcd(a,b-a) et a<b}
        finsi
    fintantque
    {a = pgcd(a,b)}
    rendre a
finfonc {pgcd}
```

L'application successive des fonctions $f(a,b) = a - b$ lorsque $a > b$ et $f'(a,b) = b - a$ lorsque $a < b$ tend vers l'égalité de a et de b. Le prédicat d'achèvement sera donc vérifié.

1. EUCLIDE, mathématicien grec du III^e^ siècle avant J.C.

8.5.5 Multiplication

L'algorithme de multiplication suivant procède par sommes successives. Le produit $x \times y$ consiste à sommer y fois la valeur x. Toutefois, on peut améliorer cet algorithme rudimentaire en multipliant x par deux et en divisant y par deux chaque fois que sa valeur est paire. Les opérations de multiplication et de division par deux sont des opérations très efficaces puisqu'elles consistent à décaler un bit vers la gauche ou vers la droite.

```
fonction produit(données x, y : naturel): naturel
{Antécédent : x = α, y = β}
{Conséquent : produit = x × y = α × β}
variable p type naturel

   p ← 0
   {α × β = p + x × y }
   tantque y>0 faire
      {α × β = p + x × y et y>0}
      tantque y est pair faire
         {α × β = p + x × y et y = (y/2) × 2 > 0}
         {α × β = p + 2x × (y/2) et y = (y/2) × 2 > 0}
         y ← y/2
         {α × β = p + 2x × y}
         x ← 2×x
         {α × β = p + x × y }
      fintantque
      {α × β = p + (x-1) × y + y et y>0 et y impair}
      p ← p+x
      {α × β = p + x × y-1 et y impair}
      y ← y-1
      {α × β = p + x × y}
   fintantque
   {y = 0 et α × β = p + x × y = p}
   rendre p
finfonc {produit}
```

La finitude de la boucle la plus interne est évidente. Les divisions entières successives par deux d'un nombre pair tendent nécessairement vers un nombre impair. La boucle externe se termine bien également puisque la fonction $f(y) = y - 1$ fait tendre y vers 0 pour tout $y > 0$.

8.5.6 Puissance

L'algorithme d'élévation à la puissance suit le même principe que précédemment. Le calcul de x^y consiste à faire y fois le produit x. De la même façon, lorsque y est pair, x est élevé au carré tandis que y est divisé par deux.

```
fonction puissance(données x, y : naturel): naturel
{Antécédent : x = α, y = β}
{Conséquent : puissance = x^y = α^β}
variable p type naturel
```

```
    p ← 1
    {α^β = p × x^y}
    tantque y>0 faire
          {α^β = p × x^y et y > 0}
       tantque y est pair faire
          {α^β = p × x^y et y = (y/2) × 2 > 0}
          y ← y/2
          {α^β = p × x^2y}
          x ← x×x
          {α^β = p × x^y}
       fintantque
       {α^β = p × x^y et y impair}
       p ← p×x
       {α^β = p × x^y-1 et y impair}
       y ← y-1
       {α^β = p × x^y}
    fintantque
    {α^β = p× x^y et y=0 ⇒ p = α^β}
    rendre p
finfonc {puissance}
```

La finitude des deux boucles se démontre comme celle des deux boucles de fonction `produit`.

8.6 EXERCICES

Exercice 8.1. Programmez en JAVA les fonctions données dans ce chapitre.

Exercice 8.2. Le calcul de factorielle est également possible en procédant de façon décroissante. Écrivez une fonction `factorielle` selon cette méthode. Vous prouverez la validité de votre algorithme, et démontrerez la finitude de la boucle.

Exercice 8.3. Montrez formellement que dans l'algorithme de la recherche du minimum et du maximum d'une suite d'entiers donnée à la page 81, l'utilisation de l'énoncé conditionnel suivant est faux.

```
si x < min alors min ← x
  sinon
    si x > max alors max ← x finsi
finsi
```

Exercice 8.4. Écrivez une fonction qui calcule le sinus d'une valeur réelle positive d'après son développement en série entière :

$$\sin(x) = \sum_{i=0}^{+\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Le calcul de la somme converge lorsqu'une précision ε est atteinte, c'est-à-dire lorsqu'un nouveau terme t est tel que $|t| \leqslant \varepsilon$. Note : ne pas utiliser la fonction factorielle.

Exercice 8.5. Les opérateurs << et >> du langage JAVA permettent de faire des décalages binaires vers la gauche ou vers la droite. L'opérande gauche est la configuration binaire à décaler, et l'opérande droit est le nombre de bits à décaler. Pour un décalage vers la gauche, des zéros sont systématiquement réinjectés par la droite. Pour un décalage vers la droite, les bits réinjectés à gauche sont des uns si la configuration binaire est négative, sinon ce sont des zéros. L'opérateur >>> décale vers la droite et réinjecte exclusivement des zéros.

À l'aide des opérateurs de décalage binaire et de l'opérateur `&` (et logique), récrivez le produit de deux entiers naturels `x` et `y`, ainsi que l'élévation de `x` à la puissance `y`. L'algorithme utilisera la décomposition binaire de `y`. On rappelle que la multiplication par deux d'un entier correspond à un décalage d'un bit vers la gauche, et que la division par deux d'un entier correspond à un décalage d'un bit vers la droite. D'autre part, l'opération `x & 1` retourne la valeur du bit le plus à droite de `x`.

Exercice 8.6. Calculez pgcd(2015,1680) à l'aide de l'algorithme donné page 82. Combien d'itérations sont nécessaires à ce calcul ? L'algorithme d'EUCLIDE s'écrit également sous la forme suivante :

```
fonction pgcd(données a, b: naturel): naturel
variable
   reste type naturel

   si a>b alors échanger(a,b) finsi
   tantque b≠0 faire
         reste ← a modulo b
         a ← b
         b ← reste
   fintantque
   rendre a
finfonc {pgcd}
```

Combien d'itérations nécessitent les calculs de pgcd(2015,1680) et pgcd(6765,10946). Prouvez la validité de cet algorithme. Comparez les deux écritures de l'algorithme. Pensez-vous que cette seconde version est plus efficace ?

Exercice 8.7. Une fraction est représentée par deux entiers naturels : le numérateur et le dénominateur. Définissez une classe `Fraction` et un méthode `irréductible` qui rend irréductible la fraction courante.

Exercice 8.8. Programmez la fonction booléenne `estPremier` qui teste si l'entier naturel passé en paramètre est un nombre premier. On rappelle qu'un nombre est premier s'il n'admet comme diviseur que 1 et lui-même. En d'autres termes, n est premier si le reste de la division entière de n par d n'est jamais nul, $\forall d, 2 \leqslant d \leqslant n-1$.

Exercice 8.9. On désire calculer la racine carrée d'un nombre réel x par la méthode de HÉRON L'ANCIEN[2]. Pour cela, on calcule la suite $r_n = (r_{n-1} + x/r_{n-1})/2$ jusqu'à obtenir

2. HÉRON L'ANCIEN, mathématicien et mécanicien grec du I^er^ siècle.

une approximation $r_n = \sqrt{x}$ telle que $|r_n^2 - x|$ est inférieur à un ε donné. Vous pourrez choisir $r_0 = x/2$. Écrivez un algorithme itératif qui procède à ce calcul.

Exercice 8.10. En utilisant la suite $r_n = (2r_{n-1} + x/(r_{n-1})^2)/3$, écrivez l'algorithme qui calcule la racine cubique d'un entier naturel x.

Chapitre 9

Les tableaux

Les tableaux définissent un nouveau mode de structuration des données. Un *tableau* est un agrégat de composants, objets élémentaires ou non, de *même* type et dont l'accès à ses composants se fait par un indice calculé.

9.1 DÉCLARATION D'UN TABLEAU

D'une façon générale, les composants d'un tableau sont en nombre fini, et sont accessibles individuellement sans ordre particulier de façon directe grâce à un *indice calculé*. Cette dernière opération s'appelle une *indexation*. La définition d'un tableau doit faire apparaître :

- le type des composants ;
- le type des indices.

La déclaration d'un tableau `t` possèdera la syntaxe suivante :

```
t type tableau [T1] de T2
```

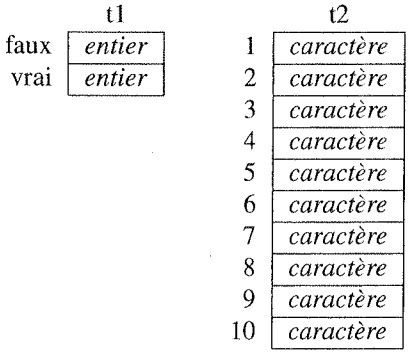
où T_1 et T_2 sont, respectivement, le type des indices et le type des composants. Il n'y a aucune restriction sur le type des composants, alors que le type des indices doit être discret. En général, tous les types simples sont autorisés (hormis le type réel) sur lesquels doit exister une relation d'ordre. Notez que cette déclaration définit une application de T_1 vers T_2.

Un tableau définit un ensemble de valeurs dont la cardinalité est égale à $| T_2 |^{|T_1|}$, où $| T |$ désigne le nombre d'éléments du type T. La cardinalité du type T_1 correspond au nombre de composants du tableau.

➢ Exemples de déclarations de tableaux

```
t1 type tableau [booléen] de entier
t2 type tableau [ [1,10] ] de caractère
```

La variable `t1` est un tableau à deux composants de type entier. Le type des indices est le type booléen. La variable `t2` un tableau à dix composants de type caractère. Le type des indices de `t2` est le type intervalle `[1,10]`. La figure suivante montre une représentation de ces deux tableaux en mémoire. Les composants sont rangés de façon contiguë. Chaque case est un composant particulier, dont le type est indiqué en italique et à gauche la valeur de l'indice qui permet d'y accéder.



Dans certains langages de programmation, comme C ou PASCAL, le nombre de composants d'un tableau est constant et figé par sa déclaration à la compilation. Au contraire, d'autres langages définissent des tableaux *dynamiques* dont le nombre d'éléments peut varier à l'exécution.

9.2 DÉNOTATION D'UN COMPOSANT DE TABLEAU

Les composants sont dénotés au moyen du nom de la variable désignant l'objet de type tableau et de l'*indice* qui désigne de façon unique le composant désiré.

On désignera un composant d'un tableau `t` d'indice `i`, par la notation `t[i]` (prononcée `t` de `i`), qui est une expression fournissant comme valeur un objet du type des composants ; `i` peut être une expression pourvu qu'elle fournisse une valeur du type des indices. La variable `t` est de type tableau alors que `t[i]` est du type des composants. Avec les déclarations précédentes, les notations suivantes sont valides :

```
t1[vrai]     t1[faux]     t1[non vrai]     {de type entier}
t2[1]        t2[10]       t2[3+4]          {de type caractère}
```

Il est important de bien comprendre que l'indice doit renvoyer une valeur sur le type des indices, ce qui n'est pas toujours facilement décelable.

```
t2[23]
t2[i+j]
```

La première notation est manifestement fausse, alors que la seconde dépend des valeurs de `i` et de `j`, qui risquent de n'être connues qu'à l'exécution du programme. Donner un indice hors de son domaine de définition est une erreur de programmation. Selon les programmes, les langages et les compilateurs, cette erreur sera signalée dès la compilation, à l'exécution, ou encore pas du tout. Cette dernière façon de procéder est celle du langage C, rendant la construction de logiciels fiables plus difficile.

9.3 MODIFICATION SÉLECTIVE

La notation `t[i]` désigne un composant particulier du tableau `t`. On peut considérer que `t[i]` est un nom de variable, et nous utiliserons cette notation pour obtenir, et surtout pour modifier de façon *sélective*, c'est-à-dire modifier un composant indépendamment des autres, la valeur de l'objet qu'elle désigne. Il sera alors possible d'écrire :

```
t1[vrai] ← 234      t1[faux] ← 0      t1[non p et q] ← -13
t2[4] ← 'z'         t2[i+5] ← 'a'     t2[i-j] ← '0'
```

9.4 OPÉRATIONS SUR LES TABLEAUX

Les langages de programmation n'offrent, en général, que peu d'opérations prédéfinies sur les tableaux. Le plus souvent, ils proposent les opérations de comparaison, qui testent si deux tableaux sont strictement identiques ou différents, ainsi que l'affectation. Deux tableaux sont égaux s'ils sont de même type, et si tous leurs composants sont égaux. L'affectation affecte individuellement chaque composant du tableau, en partie droite de l'affectation, aux composants correspondants du tableau en partie gauche.

```
variables t1, t2 type tableau[(bleu,rouge,vert)] de réel
    ...
    t2 ← t1
    ...
    si t1 = t2 alors ...
```

9.5 LES TABLEAUX EN JAVA

Une déclaration de tableau à la forme suivante :

```
Tc [] t;
```

où `Tc` est le type des composants du tableau `t`. Par exemple, les déclarations d'un tableau d'entiers `t1` et de rectangles `t2` s'écrivent :

```
int [] t1;          // un tableau d'entiers
Rectangle [] t2; // un tableau de Rectangle
```

Cette déclaration *ne crée pas* les composants du tableau. Remarquez qu'elle n'indique pas non plus leur nombre. Elle définit simplement une référence à un objet de type tableau.

La création des composants du tableau se fait *dynamiquement* à l'exécution du programme. Celle-ci doit être explicite à l'aide de l'opérateur **new** qui en précise leur nombre.

```
t1 = new int [ 10 ];
t2 = new Rectangle [ 5 ];
```

La première déclaration crée un tableau de dix éléments de type entier, et la seconde, un tableau de cinq éléments de type Rectangle. Le nombre d'éléments d'un tableau est fixé une bonne fois pour toutes lors de sa création et chaque instance de tableau possède un attribut, `length`, dont la valeur est égale au nombre d'éléments.

Une autre façon de créer les composants d'un tableau consiste à énumérer leurs valeurs lors de la déclaration :

```
int [] t1 = { 4, -5, 12 };
Rectangle [] t2 = { null, new Rectangle(3,5) };
```

Le tableau `t1` est formé de trois composants dont les valeurs sont `4`, `-5` et `12`. Le tableau `t2` possède deux composants de type `Rectangle`, le premier est la référence **`null`**, le second est une référence sur une instance particulière (*i.e.* un rectangle de largeur 3 et de longueur 5).

Le type des indices des tableaux est *toujours* un intervalle d'entiers dont la borne inférieure est zéro et la borne supérieure le nombre d'éléments moins un. Le premier élément, `t[0]`, est à l'indice 0, le deuxième, `t[1]`, à l'indice 1, *etc.* Le dernier élément est `t[t.length-1]`. Si un indice `i` n'appartient pas au type des indices d'un tableau `t`, l'erreur produite par la notation `t[i]` sera signalée à l'exécution par un message d'avertissement[1].

À la création d'un tableau, les éléments de type numérique sont initialisés par défaut à zéro, à faux lorsqu'ils sont de type booléen, et au caractère d'ordinal zéro pour le type caractère. Si ce sont des objets de type classe (ou des tableaux à leur tour), les instances de classe de chaque élément du tableau *ne sont pas créées*. La valeur de chaque élément est égal à la référence **`null`**.

Dans la mesure où une variable de type tableau est une référence à une instance de tableau, les instructions suivantes :

```
t1 = t2
t1 == t2
```

n'affectent pas les éléments de `t2` à `t1` et ne comparent pas les éléments des tableaux `t1` à `t2`. Au contraire, elles affectent et comparent les *références* à ces deux tableaux. L'affectation et la comparaison des éléments devront être programmées *explicitement* élément à élément. La méthode `clone` de la classe `Object` et celles de la classe `Arrays` aideront le programmeur dans cette tâche.

Notez que cette situation se produit également lors d'un passage d'un tableau[2] en paramètre. Seule la référence au tableau est transmise par valeur.

1. De la forme `ArrayIndexOutOfBoundsException`.

2. Et plus généralement, pour toute instance d'objet.

9.6 UN EXEMPLE

Nous voulons initialiser de façon aléatoire les composants d'un tableau d'entiers et rechercher dans le tableau l'occurrence d'une valeur entière particulière. Les déclarations suivantes définissent un tableau `tab` à n composants entiers.

```
constante n = 10 {nombre d'éléments du tableau}
variable tab type tableau [ [1,n] ] de entier
```

L'algorithme d'initialisation suivant utilise la fonction `random` qui retourne un entier tiré de façon aléatoire.

```
Algorithme Initialisation aléatoire
   {Rôle: initialise le tableau tab de façon aléatoire}
   variable i type entier
   i ← 0
   répéter
      {∀i ∈ [1,i], tab[i] est initialisé aléatoirement}
      i ← i+1
      tab[i] ← random() {random retourne un nb aléatoirement}
   jusqu'à i=n
   {∀i ∈ [1,n], tab[i] est initialisé aléatoirement}
```

L'initialisation du tableau est un parcours systématique de tous les éléments, du premier au dernier. Le nombre d'itérations est donc connu à l'avance. Remarquez que les énoncés itératifs **répéter** et **tantque** ne sont pas vraiment adaptés puisqu'ils réclament un prédicat d'achèvement. Nous verrons au prochain chapitre, comment l'énoncé itératif **pour** offre une notation simple pour l'écriture d'une telle boucle.

L'algorithme de recherche parcourt le tableau de façon linéaire à partir du premier élément. Si l'élément est trouvé, il retourne la valeur *vrai*, sinon il retourne la valeur *faux*.

```
Algorithme Recherche linéaire
   {Antécédent : x entier à rechercher dans le tableau tab}
   {Conséquent : retourne x ∈ tab}
   variable i type entier
     i ← 1
     répéter
        {∀i ∈ [1,i-1], x ≠tab[i]}
        si tab[i]=x {trouvé}
               alors rendre vrai
               sinon i ← i+1
        finsi
     jusqu'à i=n
     {∀i ∈ [1,n], x≠tab[i]}
     rendre faux
```

Programmation en JAVA

Nous allons définir une classe `TabAléa` dont le constructeur se chargera d'initialiser le tableau. Puisque JAVA permet la construction dynamique des composants du tableau, le nombre d'éléments sera passé en paramètre au constructeur.

La classe `Random`, définie dans le paquetage `java.util`, permet de fabriquer des générateurs de valeurs entières ou réelles calculées de façon pseudo-aléatoire. La méthode `nextInt` retourne le prochain entier.

```
import java.util.*;
class TabAléa {
    // Invariant de classe: TabAléa représente une suite
    // aléatoire de n entiers
    int [] tab;

    TabAléa(int n)
    // Rôle: créer les n composants du tableau tab
    // et initialiser tab de façon aléatoire
    {
        // créer un générateur de nombres aléatoires
        Random rand = new Random();
        // créer les n composants du tableau
        tab  = new int [n];
        int i=0;
        do
            // ∀i ∈ [0,i-1], tab[i] est initialisé aléatoirement
            tab[i++] = rand.nextInt();
        while (i<n);
        // ∀i ∈ [0,n-1], tab[i] est initialisé aléatoirement
    }
    boolean rechercher(int x)
    // Antécédent: x entier à rechercher
    // Conséquent: retourne x ∈ tab
    {
        int i=0;
        do
           // ∀i ∈ [0,i-1], x≠tab[i]
           if (tab[i++]==x) // trouvé
               return true;
        while (i<tab.length);
        // ∀i ∈ [0,tab.length-1], x≠tab[i]
        return false;
    }
} // fin classe TabAléa
```

L'expression `i++` incrémente la variable `i` de un, et renvoie comme résultat la valeur de `i` avant l'incrémentation. N'oubliez pas qu'en JAVA une affectation est une expression. La comparaison `(tab[i++] == x)` compare donc `tab[i]` à `x` *avant* l'incrémentation de `i`. Notez que l'expression `++i` renvoie comme résultat la valeur de `i` après incrémentation.

9.7 LES CHAÎNES DE CARACTÈRES

Dans les chapitres précédents, nous avons déjà manipulé des chaînes de caractères sans connaître exactement la nature de cet objet. Nous savons qu'une constante chaîne se dénote entre deux guillemets :

```
"toto"     "bonjour à tous"
"l'été"
```

Certains langages de programmation, comme PASCAL ou C par exemple, représentent les chaînes de caractères par des tableaux de caractères. D'autres proposent, comme SNOBOL (voir page 9), un type prédéfini chaîne de caractères sans préciser de représentation particulière. Le nombre d'opérations prédéfinies sur les chaînes varie considérablement d'un langage à l'autre. Certains n'en définissent que très peu, comme PASCAL par exemple, d'autres, au contraire comme SNOBOL, en proposent une multitude.

Outre la dénotation de constantes et la déclaration de variables, les opérations traditionnelles de manipulation de chaînes sont la concaténation, le calcul de la longueur, la recherche de caractères ou de sous-chaînes, *etc.*

➢ Les chaînes de caractères en Java

Une constante chaîne de caractères est une suite de caractères dénotée entre guillemets. Tous les caractères du jeu UNICODE sont autorisés, y compris leur représentation spéciale (voir page 28). Par exemple, la constante `"bonjour\nà␣tous␣\u2665"` dénote les mots `bonjour` et `à tous`, séparés par un passage à la ligne, et suivis du symbole ♥.

L'environnement JAVA définit deux classes, `String` et `StringBuffer`, pour créer et manipuler des chaînes de caractères. Les objets de type `String` sont des chaînes constantes, *i.e.*, une fois créées, elles ne peuvent plus être modifiées. Au contraire, les chaînes de type `StringBuffer` peuvent être modifiées dynamiquement.

Ces deux classes offrent une multitude de services que nous ne décrirons pas ici. Nous nous contenterons de présenter quelques fonctions classiques de la classe `String`.

La méthode `length` retourne le nombre de caractères contenus dans la chaîne courante. La méthode `charAt` retourne le caractère dont la position dans la chaîne courante est passée en paramètre. La position du premier caractère est zéro. La méthode `indexOf` retourne la première position du caractère passé en paramètre dans la chaîne courante.

La méthode `compareTo` compare la chaîne courante avec une chaîne passée en paramètre. La comparaison, selon l'ordre lexicographique, retourne zéro si les chaînes sont identiques, une valeur entière négative si l'objet courant est inférieur au paramètre, et une valeur entière positive si l'objet courant lui est supérieure.

De plus, le langage définit l'opérateur + qui concatène deux chaînes[3], et d'une façon générale deux objets munis de la méthode `toString`.

Le fragment de code suivant déclare deux variables de type `String` et met en évidence les méthodes données plus haut :

3. L'opérateur + possède ainsi plusieurs significations, puisque nous avons déjà vu qu'il permettait d'additionner des valeurs numériques.

```
String c1 = new String("bonjour␣"); // ou String c1 = "bonjour ";
String c2;

c2 = c1 + "à␣tous";
System.out.println(c2);
System.out.println(c2.length());
System.out.println(c2.charAt(3));
System.out.println(c2.indexOf('o'));
System.out.println(c1.compareTo("à␣tous"));
```

L'exécution de ce fragment de code produit les résultats suivants :

```
bonjour à tous
14      // longueur de la chaîne "bonjour à tous"
j       // le quatrième caractère
1       // position du premier 'o'
-126    // "bonjour "<"à tous"
```

Le résultat de la comparaison paraît surprenant, puisque la lettre *a*, même accentuée, précède la lettre *b* dans l'alphabet français. En fait, elle suit l'ordre des caractères dans le jeu UNICODE. Toutefois, JAVA définit la classe `Collator` (du paquetage `java.text`) qui connaît les règles de comparaisons spécifiques à différentes langues internationales. À la création d'une instance de type `Collator`, on indique le langage désiré, puis on utilise la méthode `compare` à laquelle on passe en paramètre les deux chaînes à comparer. Le résultat de la comparaison suit la même convention que celle de `compareTo`. La comparaison suivante retourne une valeur positive.

```
Collator fr = Collator.getInstance(Locale.FRENCH);
System.out.println(fr.compare(c1, "à␣tous"));
```

9.8 EXERCICES

Exercice 9.1. Écrivez et démontrez la validité d'une fonction qui calcule la moyenne des éléments d'un tableau d'entiers.

Exercice 9.2. Écrivez et démontrez la validité d'une fonction qui recherche la valeur minimale et la valeur maximale d'un tableau de n entiers.

Exercice 9.3. On appelle *palindrome* un mot ou une phrase qui, lu de gauche à droite ou, inversement, de droite à gauche garde le même sens. Le mot *radar* ou la phrase *esope reste et se repose* sont des palindromes. Écrivez une fonction qui teste si une chaîne de caractères est un palindrome.

Exercice 9.4. On désire rechercher tous les nombres premiers compris entre 2 et une certaine valeur maximale n, selon l'algorithme du crible d'ÉRATOSTHÈNE[4]. Le crible est la structure qui contient la suite d'entiers ; il est représenté habituellement par un tableau.

4. Mathématicien et philosophe, connu pour ses travaux en arithmétique et en géométrie, ÉRATOSTHÈNE vécut au III^{e} siècle avant J.C. à Alexandrie.

Écrivez et démontrez la validité d'une procédure qui affiche sur la sortie standard les nombres premiers compris entre 2 et n selon l'algorithme :

```
Algorithme Crible d'ÉRATOSTHÈNE
    initialiser le crible
    vide ← faux
    répéter
        {le plus petit nombre contenu dans le crible est premier}
        - afficher ce nombre sur la sortie standard
        - l'enlever du crible avec tous ses multiples
        si le crible est vide alors
            vide ← vrai
        finsi
    jusqu'à vide
```

Notez que les éléments du tableau peuvent être simplement des booléens, puisque seule la présence ou l'absence du nombre dans le crible est significative.

Exercice 9.5. On cherche à définir une classe pour représenter des vecteurs de la forme $v = [x_1, x_2, x_3, \ldots, x_n]$. Les composantes réelles de chaque vecteur seront représentées par un tableau de réels. La *dimension* du vecteur, c'est-à-dire le nombre d'éléments du tableau, est fixée par le constructeur de la classe.

Écrivez en JAVA une classe `Vecteur` avec un constructeur dont le paramètre est la dimension du vecteur. La dimension sera une caractéristique de chaque objet de type `Vecteur` créé.

Écrivez et démontrez la validité de la méthode `norme` qui retourne la norme d'un vecteur. Puis, ajoutez une méthode `normalise` pour normaliser les composantes du vecteur courant. On rappelle que la norme d'un vecteur v est égale à :

$$\|v\| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

et que la normalisation d'un vecteur est donnée par :

$$\bar{v} = \frac{v}{\|v\|} = \left[\frac{x_1}{\|v\|}, \frac{x_2}{\|v\|}, \ldots, \frac{x_n}{\|v\|}\right]$$

Écrivez et démontrez la validité de la méthode `produitScalaire` qui retourne le produit scalaire du vecteur transmis en paramètre et du vecteur courant. On vérifiera que les deux vecteurs possèdent bien la même dimension. On rappelle que le produit scalaire de deux vecteurs v et v' est égal à :

$$v.v' = \sum_{i=1}^{n} x_i \, x'_i$$

Chapitre **10**

L'énoncé itératif `pour`

10.1 FORME GÉNÉRALE

Il arrive que l'on ait besoin de faire le même traitement sur toutes les valeurs d'un type donné. Par exemple, on désire afficher tous les caractères contenus dans le type caractère du langage de programmation avec lequel on travaille. Beaucoup de langages de programmation proposent une construction adaptée à ce besoin spécifique, appelée énoncé itératif **`pour`**. Une forme générale de cette construction est un énoncé qui possède la syntaxe suivante :

```
pourtout x de T faire E finpour
```

où x est une variable de boucle qui prendra successivement toutes les valeurs du type `T`. Pour chacune d'entre elles, l'énoncé `E` sera exécuté. Notez, d'une part, que l'ordre de parcours des éléments du type T n'a pas nécessairement d'importance, et d'autre part, que la variable de boucle n'est définie et n'existe, qu'au moment de l'exécution de l'énoncé itératif.

L'énoncé suivant écrit sur la sortie standard tous les caractères du type caractère.

```
pourtout c de caractère faire
  écrire(c)
finpour
```

Il est important de comprendre que le nombre d'itérations ne dépend pas d'un prédicat d'achèvement, contrairement aux énoncés **`tantque`** ou **`répéter`**. Il est égal au cardinal du type `T`. On a la garantie que la boucle s'achève et la finitude de la boucle n'est donc plus à démontrer ! Dans un algorithme, chaque fois que vous aurez à effectuer des itérations dont le nombre peut être connu à l'avance de *façon statique*, vous utiliserez l'énoncé **`pourtout`**.

10.2 FORME RESTREINTE

La plupart des langages de programmation définissent des formes restreintes de l'énoncé général **pourtout**. En particulier, elles limitent le type T aux types élémentaires discrets, et fixent un ordre de parcours sur un intervalle [min , max]. Cet ordre peut être croissant ou décroissant selon que la borne minimale est un élément de valeur inférieure ou supérieure à celle de la borne maximale. Nous dénoterons cette forme restreinte comme suit :

```
pourtout x de min à max faire E finpour
```

Avec cet énoncé, l'algorithme d'initialisation d'un tableau, donné à la page 91, s'écrit simplement :

```
Algorithme Initialisation aléatoire
    {Rôle: initialise le tableau tab de façon aléatoire}
    pourtout i de 1 à n faire
        {∀i ∈ [1,i-1], tab[i] est initialisé aléatoirement}
        {random retourne un nombre de façon aléatoire}
        tab[i] ← random()
    finpour
    {∀i ∈ [1,n], tab[i] est initialisé aléatoirement}
```

➢ Règles de déduction

La règle de déduction de l'énoncé pour restreint dans le cas d'un parcours croissant des valeurs de l'intervalle est donné ci-dessous. La fonction *pred* retourne le prédécesseur d'un élément dans l'intervalle `]min,max]`.

$$\underline{\text{si}}\ \{v = min\ et\ P\} \overset{E_v}{\Rightarrow} \{Q_{v_{min}}\}\ \underline{\text{et}}\ \{Q_{pred(v_x)}\} \overset{E_{v_x}}{\Rightarrow} \{Q_{v_x}\}\ \forall x \in\ ]min,max]$$
$$\underline{\text{alors}}$$
$$\{min \leqslant max\ et\ P\}\ \textbf{pourtout}\ \texttt{v}\ \textbf{de}\ \texttt{min}\ \textbf{à}\ \texttt{max}\ \textbf{faire}\ \texttt{E}\ \textbf{finpour}\ \{Q_{v_{max}}\}$$

De la même manière, la règle de déduction pour un parcours décroissant de l'intervalle est le suivant. La fonction *succ* retourne le successeur d'un élément dans l'intervalle $[min,max[$.

$$\underline{\text{si}}\ \{v = max\ et\ P\} \overset{E_v}{\Rightarrow} \{Q_{v_{max}}\}\ \underline{\text{et}}\ \{Q_{succ(v_x)}\} \overset{E_{v_x}}{\Rightarrow} \{Q_{v_x}\}\ \forall x \in\ [min,max[$$
$$\underline{\text{alors}}$$
$$\{min \leqslant max\ et\ P\}\ \textbf{pourtout}\ \texttt{v}\ \textbf{de}\ \texttt{max}\ \textbf{à}\ \texttt{min}\ \textbf{faire}\ \texttt{E}\ \textbf{finpour}\ \{Q_{v_{min}}\}$$

10.3 L'ÉNONCÉ POUR DE JAVA

La sémantique de l'énoncé **pour** de JAVA n'a malheureusement pas le caractère fondamental des énoncés précédents, puisque le prédicat d'achèvement apparaît dans sa notation :

```
for (exp1; exp2; exp3) E
```

où exp1 est une expression de déclaration et d'initialisation de la variable de boucle, exp2 est le prédicat d'achèvement, et exp3 est l'expression d'incrémentation de la variable de boucle. Cet énoncé est *strictement* équivalent à l'énoncé **tantque** suivant :

```
exp1;
while (exp2) { E; exp3; }
```

Cette forme d'énoncé ne dispensera donc pas le programmeur de démontrer la finitude de la boucle. Le constructeur de la classe TabAlea de la page 92 s'écrit avec cet énoncé :

```
TabAléa(int n)
 // Rôle: créer les n composants du tableau tab
 // et les initialiser de façon aléatoire
 {
     // créer un générateur de nombres aléatoires
     Random rand = new Random();
     // créer les n composants du tableau
     tab=new int [n];
     for (int i=0; i <n; i++)
         // ∀i ∈ [0,i-1], tab[i] est initialisé aléatoirement
         tab[i]=rand.nextInt();
     // ∀i ∈ [0,n-1], tab[i] est initialisé aléatoirement
}
```

10.4 EXEMPLES

10.4.1 Le schéma de HORNER

Nous voulons calculer la valeur d'un polynôme p de degré n à une variable représenté par le tableau de ses coefficients :

```
cf type tableau [ [0,n] ] de réels
```

Une valeur de p pour une variable x est donnée par :

$$p(x) = \texttt{cf[0]} \times x^0 + \texttt{cf[1]} \times x^1 + \texttt{cf[2]} \times x^2 + \cdots + \texttt{cf[n]} \times x^n$$

Les élévations à la puissance successives rendent le calcul de $p(x)$ très coûteux. Pour éviter les calculs successifs des x^i, on utilise le schéma de HORNER [1], donné ci-dessous :

$$p(x) = (((\ldots(\texttt{cf[n]} \times x + \texttt{cf[n-1]}) \times x + \cdots + \texttt{cf[1]}) + \texttt{cf[0]}$$

Par exemple, le polynôme $3x^3 - 2x^2 + 4x + 1$ est « récrit » sous la forme $((3x - 2)x + 4)x + 1$. L'invariant de l'algorithme du calcul de la valeur $p(x)$ par le schéma de HORNER, $valeur = \sum_{k=n}^{i} cf[k]\, x^{k-i}$, se démontre aisément par application de la règle de déduction donnée plus haut.

1. Redécouverte au début du XIX^{e} siècle par l'anglais W. HORNER, cette méthode est due au mathématicien chinois CHU SHIH-CHIEH, 500 ans plus tôt.

Algorithme HORNER

```
{Rôle : calcul de la valeur d'un polynôme
       de degré n à une variable
       x et le tableau des coefficients cf}
valeur ← cf[n]
{valeur = ∑_{k=n}^{n} cf[k]x^{k-n} = cf[n]}
pourtout i de n-1 à 0 faire
    {valeur = ∑_{k=n}^{i} cf[k]x^{k-i}}
    valeur ← valeur × x + cf[i]
finpour
{valeur = ∑_{k=n}^{0} cf[k]x^{k}}
rendre valeur
```

Programmation en JAVA

La programmation en JAVA de l'algorithme précédent est donnée ci-dessous. On considère que le tableau `cf` est un attribut de la classe dans laquelle la fonction `Horner` a été définie.

```
public double Horner(double x) {
    // Rôle: calcul de la valeur d'un polynôme
    // de degré n à une variable
    int n=cf.length-1;
    double valeur=cf[n];
    // valeur = ∑_{k=n}^{n} cf[k]x^{k-n} = cf[n]
    for (int i=n-1; i>=0; i--)
        // valeur = ∑_{k=n}^{i} cf[k]x^{k-i}
        valeur*=x+cf[i];
    // valeur = ∑_{k=n}^{0} cf[k]x^{k}
    return valeur;
}
```

10.4.2 Un tri interne simple

Une primitive de tri consiste à ordonner, de façon croissante ou décroissante, une liste d'éléments. Par exemple, si nous considérons la liste de valeurs entières suivante :

```
53  914  827  302  631  785  230   11  567  350
```

une opération de tri ordonnera ces valeurs de façon croissante et retournera la liste :

```
11   53  230  302  350  567  631  785  827  914
```

Nous présentons une méthode de tri simple, appelée tri par sélection ordinaire. Ce tri est dit interne car l'ensemble des éléments à trier réside en mémoire principale, dans un *tableau*. Il existe de nombreuses méthodes de tri, plus ou moins performantes, que nous étudierons en détail au chapitre 22 page 295.

Le principe de la sélection ordinaire est de rechercher le minimum de la liste, de le placer en tête de liste et de recommencer sur le reste de la liste. En utilisant la liste d'entiers précédente, le déroulement de cette méthode donne les étapes suivantes. À chaque étape, le minimum recherché est souligné.

```
| 53  914  827  302  631  785  230   11  567  350
11  | 914  827  302  631  785  230   53  567  350
11   53  | 827  302  631  785  230  914  567  350
11   53  230  | 302  631  785  827  914  567  350
11   53  230  302  | 631  785  827  914  567  350
11   53  230  302  350  | 785  827  914  567  631
11   53  230  302  350  567  | 827  914  785  631
11   53  230  302  350  567  631  | 914  785  827
11   53  230  302  350  567  631  785  | 914  827
11   53  230  302  350  567  631  785  827  | 914
```

L'algorithme de tri suit un processus itératif dont l'invariant spécifie, d'une part, qu'à la ième étape la sous-liste formée des éléments de `t[1]` à `t[i-1]` est triée, et d'autre part, que tous ses éléments sont inférieurs ou égaux aux éléments `t[i]` à `t[n]`. On en déduit que le nombre d'étapes est `n-1`.

Algorithme `Tri par sélection ordinaire`

```
{Rôle : Trie par sélection ordinaire en ordre croissant}
{       les n valeurs d'un tableau t}
pourtout i de 1 à n-1 faire
    {Invariant : le sous-tableau de t[1] à t[i-1] est trié
                 et ses éléments sont inférieurs ou égaux
                 aux éléments t[i] à t[n]}
     min ← i
     {chercher l'indice du minimum sur l'intervalle [i,n]}
     pourtout j de i+1 à n faire
         si t[j] < t[min] alors min ← j finsi
     finpour
     {échanger t[i] et t[min]}
     t[i] ↔ t[min]
finpour
{le tableau de t[1] à t[n] est trié}
```

Programmation en JAVA

La procédure suivante programme l'algorithme de tri par sélection ordinaire. Remarquez la déclaration de la variable `min` dans le corps de seconde boucle **`for`**.

```
public void sélectionOrdinaire(int [] t) {
    // Rôle: Trie par sélection ordinaire en ordre croissant
    // les valeurs du tableau t
    for (int i=0; i<t.length - 1; i++) {
        // Invariant: le sous-tableau de t[0] à t[i-1] est trié
        // et ses éléments sont inférieurs ou égaux
```

```
        // aux éléments t[i] à t[t.length-1]
        int min=i;
        // chercher l'indice du minimum sur l'intervalle [i,t.length]
        for (int j=i; j<t.length; j++)
                if (t[j]<t[min]) min=j;
        // échanger t[i] et t[min]
        int aux=t[i]; t[i]=t[min]; t[min]=aux;
    }
    // le tableau de t[0] à t[t.length-1] est trié
}
```

10.4.3 Confrontation de modèle

La confrontation de modèle (en anglais « pattern matching ») est un problème classique de manipulation de chaînes de caractères. Il s'agit de rechercher la position *pos* d'une occurrence d'un mot (le modèle) de longueur *lgmot* dans un texte de longueur *lgtexte*. Le texte et le mot sont formés de caractères pris dans un alphabet Σ. L'idée générale, est de comparer répétitivement le mot à une portion du texte, appelée *fenêtre*, possédant le même nombre de caractères que le mot recherché. Une occurrence est trouvée lorsque la fenêtre et le mot sont identiques.

La première méthode qui vient immédiatement à l'esprit est de comparer le mot à toutes les fenêtres possibles du texte, en commençant par le premier caractère du texte, puis le second, *etc.* jusqu'à trouver une concordance entre le mot et une fenêtre. Cet algorithme est donné ci-dessous. Notez qu'il ne dépend pas de l'alphabet Σ. La fonction `égal` retourne la valeur *vrai* si le mot est égal à la fenêtre de position `pos` dans le texte.

Algorithme naïf
```
pos ← 1
tantque pos < lgtexte-lgmot faire
    si égal(mot, texte, pos) alors
        {le mot est à la position pos dans le texte}
        rendre pos
    sinon
        {déplacer la fenêtre d'une position}
        pos ← pos+1
    finsi
fintantque
{le mot n'a pas été trouvé dans le texte}
```

Cette méthode n'est pas très efficace car elle teste *toutes* les positions possibles du mot dans le texte. Une seconde méthode beaucoup plus efficace, proposée par BOYER et MOORE, exploite deux idées[2] :

1. On peut comparer un mot à une fenêtre en commençant par les caractères situés à leurs extrémités.

2. Il en existe une troisième qui tient compte de suffixes déjà reconnus dans le mot, mais qui ne sera pas évoquée ici.

2. Après un échec, au lieu d'avancer d'un caractère, il est possible de faire un saut plus important. Si le dernier caractère testé dans la fenêtre n'est pas présent dans le mot, on peut déplacer la fenêtre immédiatement après ce caractère. Sinon, on fait le plus *court* déplacement qui aligne le dernier caractère testé de la fenêtre avec un caractère identique du mot.

Si, par exemple, nous recherchons le mot *noir* dans le texte *anoraks noirs*, les différentes comparaisons et les déplacements d produits lors des échecs sont donnés ci-dessous :

```
a n o r a k s     n o i r s
      :   :       :
n o i r   :       :                    échec ⇒ d = 1
    n o i r       :                    échec ⇒ d = 4
          n o i r :                    échec ⇒ d = 3
                  n o i r              succès
```

Après le premier échec, l'alignement des deux lettres *o* provoque un déplacement d'un caractère. Après le deuxième échec, et puisqu'il n'y a pas de *a* dans le mot, la fenêtre est placée immédiatement après cette lettre. La comparaison entre le *n* et le *r* échoue. La lettre *n* est présente dans le mot, l'ajustement produit un déplacement de trois caractères. Enfin, le mot et la fenêtre sont identiques.

L'algorithme donné ci-dessous retourne l'indice dans le texte du premier caractère du mot recherché s'il est trouvé, sinon il retourne la valeur -1. Nous représentons le mot et le texte par deux tableaux de caractères. La variable `pos` indique la position courante dans le texte. Nous traiterons le calcul de la valeur du déplacement plus loin.

```
Algorithme Boyer-Moore
   variables pos, i, j de type naturel
   variable différent de type booléen

   pos ← 1
   tantque pos ⩽ lgtexte-lgmot+1 faire
      différent ← faux
      { On compare à partir de la fin du mot et de la fenêtre}
      i ← lgmot
      j ← pos+lgmot-1
      répéter
            si mot[i]=texte[j] alors
                  {égalité ⇒ on poursuit la comparaison}
                  i ← i-1
                  j ← j-1
            sinon {différence ⇒ on s'arrête}
                   différent ← vrai
            finsi
      jusqu'à i=0 ou différent
      si i=0 alors {la comparaison a réussi, retourner la position}
          rendre pos
```

```
    sinon {échec: déplacer la fenêtre vers la droite}
          pos ← pos + {valeur du déplacement}
    finsi
  fintantque
  {le mot n'a pas été trouvé dans le texte}
  rendre -1
```

Lorsque la comparaison entre le mot et la fenêtre courante a échoué, c'est-à-dire lorsque `mot[i]≠texte[j]`, il faut déplacer la fenêtre vers la droite. Une façon de procéder est de chercher un caractère `mot[k]` égal au caractère `texte[j]`, et de produire le saut qui les place l'un en face de l'autre face. La figure 10.1 montre les trois cas qui peuvent se présenter : (1) ce caractère n'est pas présent dans le mot ; (2) il apparaît dans le mot avant `texte[j]` ; (3) il apparaît dans le mot après `texte[j]`.

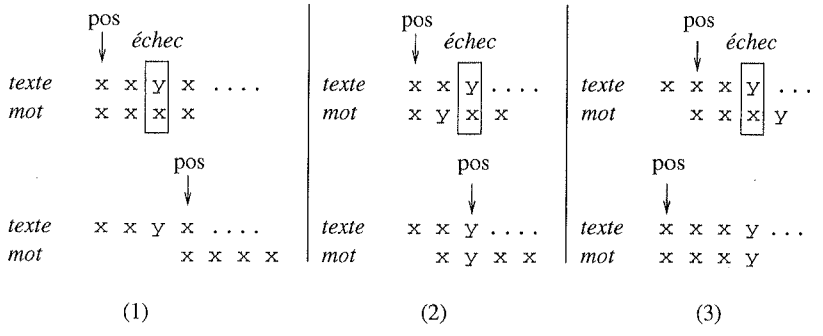


Figure 10.1 – *Déplacements après échec de la comparaison.*

Il est important de noter que le calcul du déplacement ne dépend que du mot, mais nécessite la connaissance de l'alphabet Σ. On pose $ts(c)$ l'indice du caractère c le plus à droite dans mot ; si $c \notin mot$ alors $ts(c) = 0$. La valeur du déplacement à produire est alors égale à $i - ts(texte[j])$. Notez que cette valeur peut être inférieure ou égale à zéro (troisième cas). Plutôt que de provoquer un retour en arrière, on décale la fenêtre vers la droite d'un caractère.

➢ Programmation en JAVA

Nous représenterons les variables `mot` et `texte` par deux chaînes de caractères de type `String`. La fonction ts est simplement représentée par un tableau `ts` qui possède un nombre de composants égal au cardinal de l'alphabet utilisé. Nous considérerons les 256 premiers caractères du jeu UNICODE. Notez que le tableau est initialisé à -1 pour les caractères qui n'appartiennent pas au mot. Rappelez-vous que l'indice du premier élément d'un tableau en JAVA est égal à 0. La programmation de l'algorithme de BOYER–MOORE est la suivante :

```
int BoyerMoore(String texte, String mot) {
// Rôle: recherche la première occurrence du mot dans le texte
// et retourne sa position dans le texte ou -1 si non trouvée
    int [] ts = new int[MAX_CAR];
    // initialisation de la table ts
```

```
        for (int i=0; i<MAX_CAR; i++) ts[i]=-1;
        for (int i=0; i<mot.length(); i++) ts[mot.charAt(i)]=i;
        // rechercher l'occurrence du mot
        int pos=0;
        while (pos <= texte.length()-mot.length()) {
            // on compare à partir de la fin du mot et la fenêtre
            int i=mot.length()-1;
            int j=pos+i;
            while (i>=0 && mot.charAt(i)==texte.charAt(j)) {
                    // égalité =>on poursuit la comparaison
                    i--; j--;
            }
            if (i<0) // la comparaison a réussi, retourner pos
                 return pos;
             else  // échec: déplacer la fenêtre vers la droite
                pos+=Math.max(1,i-ts[texte.charAt(j)]);
        }
        return -1;
}
```

10.5 COMPLEXITÉ DES ALGORITHMES

Nous venons de mentionner qu'il existe de nombreuses méthodes de tri plus ou moins performantes, ou encore que la méthode de confrontation de modèle de BOYER–MOORE est plus efficace que celle qui consiste à comparer le mot à toutes les fenêtres possibles. Mais à quoi correspond cette notion de performance ou d'efficacité et comment l'analyser ?

Lorsqu'on évalue les performances d'un programme, on s'intéresse principalement à son *temps d'exécution* et à la *place en mémoire* qu'il requiert. Dire, par exemple, que tel programme trie 1 000 éléments en 0,1 seconde sur telle machine et utilise quatre méga-octets en mémoire centrale n'a toutefois que peu de signification. Les caractéristiques des ordinateurs, mais aussi des langages de programmation et des compilateurs qui les implantent, sont trop différentes pour tirer des conclusions sur les performances d'un programme à partir de mesures absolues obtenues dans un environnement particulier. Mais surtout, cela ne nous permet pas de prévoir le temps d'exécution et l'encombrement mémoire de ce même programme pour le tri de 100 000 éléments. Va-t-il réclamer 100 fois plus de temps et de mémoire ? L'estimation du temps d'exécution ou de l'encombrement en mémoire d'un programme est fondamentale. Si l'on peut prédire qu'en augmentant ses données d'un facteur 100, un programme s'exécutera en trois mois ou nécessitera dix giga-octets de mémoire, il sera certainement inutile de chercher à l'exécuter.

Plutôt que donner une mesure absolue des performances d'un programme, nous donnerons une mesure théorique de son algorithme, appelée *complexité* , indépendante d'un environnement matériel et logiciel particulier. Cette mesure sera fonction d'éléments caractéristiques de l'algorithme et on supposera que chaque opération de l'algorithme prend un temps unitaire. Cette mesure ne nous permet donc pas de prévoir un temps d'exécution exact, mais ce qui nous intéresse vraiment c'est l'ordre de grandeur de l'évolution du temps d'exécution (ou de l'encombrement mémoire) en fonction d'éléments caractéristiques de l'algorithme.

Par exemple, dans le cas des tris, la mesure correspond au nombre de comparaisons d'éléments (parfois, on considère aussi le nombre de transferts) exprimé en fonction du nombre d'éléments à trier. À l'étape i du tri par sélection de n éléments (algorithme donné à la page 101), la comparaison est exécutée par la boucle la plus interne $n - i$ fois. Puisqu'il y a $n - 1$ étapes, la comparaison est donc exécutée $\sum_{i=1}^{n-1} i = \frac{1}{2}(n^2 - n)$. La complexité temporelle du tri par sélection est dite quadratique ou $\mathcal{O}(n^2)$. Cela signifie que si on multiple par 100 le nombre d'éléments à trier, on peut alors prédire que le temps d'exécution du tri sera multiplié par 10 000. La complexité spatiale du tri par sélection est linéaire $\mathcal{O}(n)$. Si on multiple par 100 le nombre d'éléments à trier, le programme utilisera un tableau 100 fois plus grand.

Quelle est la signification de la notation $\mathcal{O}(n^2)$ utilisée plus haut et pourquoi ne considère-t-on que le terme n^2 alors que le nombre exact de comparaisons est $\frac{1}{2}(n^2 - n)$?

Soient deux fonctions positives f et g, on dit que $f(n)$ est $\mathcal{O}(g(n))$ s'il existe deux constantes positives c et n_0 telles que $f(n) \leqslant cg(n)$ pour tout $n \geqslant n_0$. L'idée de cette définition est d'établir un ordre de comparaison relatif entre les fonctions f et g. Elle indique qu'il existe une valeur n_0 à partir de laquelle $f(n)$ est toujours inférieur ou égal à $cg(n)$. On dit alors que $f(n)$ est de *l'ordre de* $g(n)$. La figure 10.2 donne une illustration graphique de cette définition.

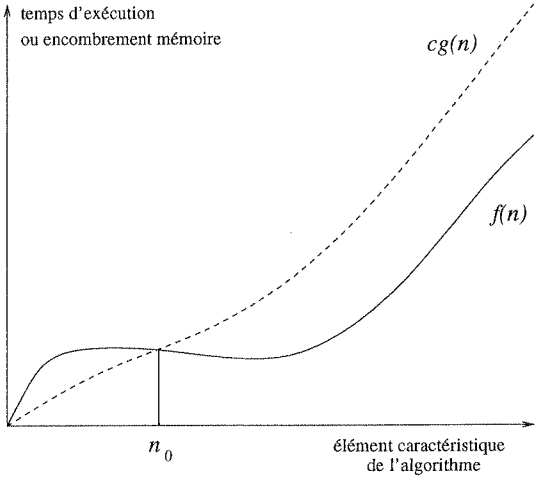


Figure 10.2 – *Représentation graphique de* $f(n) = \mathcal{O}(g(n))$.

Montrons que la fonction $\frac{1}{2}(n^2 - n)$ est de l'ordre de $\mathcal{O}(n^2)$. La définition de la notation $\mathcal{O}$ nous invite à rechercher deux valeurs positives c et n_0 telles que $\frac{1}{2}(n^2 - n) \leqslant cn^2$. En prenant par exemple $c = \frac{1}{2}$, il est évident que n'importe quel $n_0 > 0$ vérifie l'inégalité. Remarquez que nous aurions pu tout aussi bien écrire que $\frac{1}{2}(n^2 - n)$ est $\mathcal{O}(n^3)$ ou $\mathcal{O}(n^5)$, mais $\mathcal{O}(n^2)$ est plus précis. D'une façon générale, dans la notation $f(n) = \mathcal{O}(g(n))$, on choisira une fonction g la plus proche possible de f, en respectant les règles suivantes :

- $cf(n) = \mathcal{O}(f(n))$ pour tout facteur constant c.
- $f(n) + c = \mathcal{O}(f(n))$ pour tout facteur constant c.

- $f(n) + g(n) = \mathcal{O}(\max(f(n),g(n)))$.
- $f(n) \times g(n) = \mathcal{O}(f(n) \times g(n))$.
- si $f(n)$ est un polynôme de degré m, $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_m n^m$, alors $f(n)$ est de l'ordre du degré le plus grand, *i.e.* $\mathcal{O}(n^m)$.
- $n^m = \mathcal{O}(c^n)$ pour tout $m > 0$ et $c > 1$.
- $\log n^m = \mathcal{O}(\log n)$ pour tout $m > 0$.
- $\log n = \mathcal{O}(n)$

En appliquant ces règles, on en déduit facilement que $n^6 + 3n^3 - 5$ est de l'ordre de $\mathcal{O}(n^6)$, ou encore que $3n^3 + n \log n$ est de l'ordre de $\mathcal{O}(n^3)$. Le tableau suivant présente des fonctions, et les termes employés pour les désigner, qui interviendront souvent dans l'analyse de la complexité des algorithmes que nous étudierons par la suite.

1	constante
$\log_2 n$	logarithmique
n	linéaire
n^2	quadratique
n^3	cubique
c^n $(\forall c > 1)$	exponentielle

L'intérêt de la notation $\mathcal{O}$ est d'être un véritable outil de comparaison des algorithmes. Par exemple, si, pour effectuer un tri, nous devons choisir entre le tri par sélection dont la complexité, nous venons de le voir, est $\mathcal{O}(n^2)$ et le tri en tas (voir la section 22.2.2, page 298) de complexité $\mathcal{O}(n \log_2 n)$, notre choix se portera sur le premier parce que n est petit et que le tri par sélection est simple à mettre en œuvre, ou alors sur le second parce que le nombre d'éléments à trier est tel qu'il rend le premier algorithme inutilisable.

10.6 EXERCICES

On désire effectuer des calculs sur des entiers naturels trop grands pour être représentés à l'aide du type prédéfini `entier`. On définit pour cela un type nouveau `GrandEntierNat`. Un grand entier naturel est divisé en chiffres, exprimés dans une base b. Les chiffres sont mémorisés dans un tableau, de telle façon que la valeur d'un nombre a de n chiffres est égale à :

$$a = \sum_{i=0}^{n-1} \mathit{chiffres}[i] \times b^i$$

Exercice 10.1. Définissez la classe `GrandEntierNat` avec les attributs nécessaires à la représentation d'un grand entier.

Exercice 10.2. Écrivez quatre constructeurs qui initialisent un grand entier, respectivement, à zéro, à la valeur d'un entier passé en paramètre, à la valeur entière définie par une chaîne de caractères passée en paramètre, à la valeur d'un `GrandEntierNat` passé en paramètre.

Exercice 10.3. Écrivez les quatre opérations élémentaires, addition, soustraction, multiplication et division.

Exercice 10.4. Écrivez les fonctions de comparaison qui testent si deux grands entiers sont inférieurs, inférieurs ou égaux, supérieurs, supérieurs ou égaux, égaux ou différents.

Exercice 10.5. Écrivez une fonction factorielle et calculez 32!

Exercice 10.6. Calculez la somme $\sum_k 1/k!$ pour k variant de 1 à 32. Cette somme est égale à la base e des logarithmes népériens.

Exercice 10.7. Donnez la notation $\mathcal{O}$ des fonctions $n^3 \log n + 5$, $2n^{5/2}$ et 2^n.

Exercice 10.8. Comment qualifiez-vous la fonction $n \log_2 n$? linéaire ou logarithmique?

Exercice 10.9. Montrez que 2^{n+2} est $\mathcal{O}(2^n)$ et que $(n+2)^4$ est $\mathcal{O}(n^4)$.

Exercice 10.10. Quelles sont les complexités temporelles et spatiales de l'algorithme de calcul de la valeur d'un polynôme selon le schéma de HORNER?

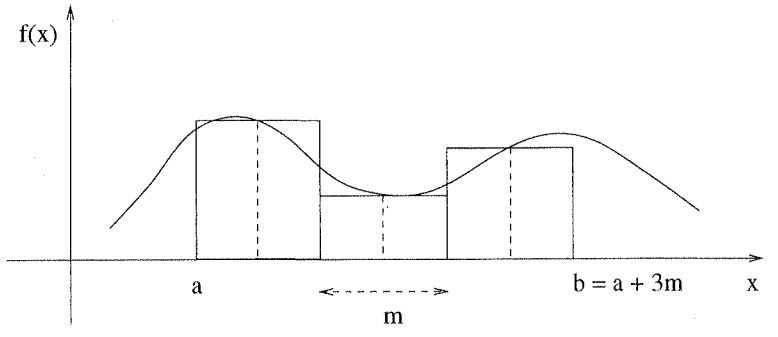
Exercice 10.11. Recherchez la k^{e} plus grande valeur d'une suite quelconque d'entiers lus sur l'entrée standard. Proposez deux algorithmes en $\mathcal{O}(n^2)$. Notez qu'il existe une méthode dont la complexité est $\mathcal{O}(n \log_2 n)$ (voir le chapitre 21).

Exercice 10.12. Dans l'analyse d'un algorithme, on est bien souvent amené à distinguer trois complexités. La complexité la meilleure, lorsque les données sont telles que l'algorithme a les meilleures performances, la complexité la pire, dans le cas défavorable où les données conduisent aux performances les moins bonnes, et enfin la complexité moyenne dans la cas général. Donnez les trois complexités temporelles de l'algorithme de BOYER–MOORE.

Exercice 10.13. La *méthode des rectangles* permet d'approximer l'intégrale d'une fonction f continue. Cette méthode consiste à découper en n intervalles $[a_i,a_{i+1}]$, de longueur identique m, l'intervalle $[a,b]$ sur lequel on veut intégrer la fonction f, puis à additionner l'aire des rectangles de hauteur $f((a_i + a_{i+1})/2)$ et de largeur m. L'aire A, approximation de l'intégrale de f sur l'intervalle $[a,b]$, vaut donc :

$$A = \sum_{i=1}^{n} m \times f((a_i + a_{i+1})/2)$$

La figure suivante illustre la méthode des rectangles dans le cas où $n = 3$.



Programmez en JAVA une méthode qui calcule l'intégrale de la fonction *cosinus* sur un intervalle $[a,b]$. Testez votre méthode, par exemple, sur l'intervalle $[0,\pi/2]$; quel est le nombre d'intervalles nécessaires pour obtenir un résultat exact à la cinquième décimale ?

Exercice 10.14. La méthode de SIMPSON fournit une approximation du calcul de l'intégrale bien meilleure que la méthode des rectangles. Elle consiste à calculer l'aire :

$$A = \sum_{i=1}^{n} m/6 \times (f(a_i) + 4 \times f(a_i + m/2) + f(a_{i+1}))$$

Programmez la méthode de SIMPSON et comparez-la de façon expérimentale avec la méthode des rectangles.

Chapitre **11**

Les tableaux à plusieurs dimensions

Les composants d'un tableau peuvent être de type quelconque et en particulier de type tableau. Les tableaux de tableaux sont souvent appelés tableaux à *plusieurs dimensions*. Certains langages de programmation, comme FORTRAN ou ALGOL 68, ont une vision différente de cette notion, mais la plupart des langages de programmation actuels se conforment à ce modèle[1].

En général, le nombre de dimensions n'est pas limité, mais dans la pratique les programmes utilisent le plus souvent des tableaux à deux dimensions et beaucoup plus rarement à trois dimensions. Un tableau à deux dimensions permet de représenter la notion mathématique de matrice.

11.1 DÉCLARATION

La déclaration d'un tableau à deux dimensions possède la forme suivante :

```
t type tableau [T1, T2] de T3
```

Bien sûr, `t` est un tableau à deux dimensions à condition que T_3 ne soit pas le type d'un tableau.

La déclaration d'une matrice de réels qui possèdent `m` lignes et `n` colonnes est donnée par :

1. Le langage PASCAL a été le premier à la fin des années 60 à proposer le modèle de tableaux de tableaux pour représenter les tableaux à plusieurs dimensions.

```
constantes
   m = 10
   n = 20
variable
   matrice type tableau [ [1,m] , [1,n] ] de réel
```

D'une façon générale, on pourra déclarer un tableau à n dimensions de la façon suivante :

```
t type tableau [T1, T2, ..., Tn] de Tc
```

où T_c est le type des composants du tableau à n dimensions.

```
constante
    n = 10
variable
    table type tableau [caractère, booléen, [1,n]] de réel
```

Dans la déclaration précédente, les types des indices de la première, de la deuxième et de la troisième composante sont, respectivement, caractère, booléen et intervalle. Le type des composants de `table` est le type réel.

11.2 DÉNOTATION D'UN COMPOSANT DE TABLEAU

Comme pour un tableau à une dimension, les composants sont dénotés au moyen du nom de la variable désignant l'objet de type tableau et d'*indices* qui désignent de façon unique le composant désiré. L'ordre des indices est celui défini par la déclaration.

```
matrice[1] {composant de type tableau [[1,n]] de réel}
matrice[1,4] {composant de type réel}

table['f'] {composant de type tableau [booléen, [1,n]] de réel}
table['f',vrai] {composant de type tableau [[1,n]] de réel}
table['f',vrai,3] {composant de type réel}
```

Les indices sont des expressions dont les résultats des évaluations doivent appartenir au type de l'indice associé.

11.3 MODIFICATION SÉLECTIVE

Les remarques faites sur la modification sélective d'un composant d'un tableau à une dimension (voir la section 9.3 page 89) s'appliquent de façon identique à un composant de tableau à plusieurs dimensions. Le fait qu'un composant de tableau soit lui-même de type tableau n'introduit aucune règle particulière.

11.4 OPÉRATIONS

Les opérations sur les tableaux à plusieurs dimensions sont identiques à celles sur les tableaux à une dimension (voir la section 9.4 page 89).

11.5 TABLEAUX À PLUSIEURS DIMENSIONS EN JAVA

Les tableaux à plusieurs dimensions sont traités comme des tableaux de tableaux. Le nombre de dimensions peut être quelconque et les règles pour leur déclaration et leur création sont semblables à celles données dans la section 9.5 page 89. On déclare un tableau `t` à n dimensions dont les composants sont de type T_c de la façon suivante :

```
Tc [][][] ... [] t;
```

La création des composantes du tableau `t` est explicitée à l'aide de l'opérateur **new**. Pour chacune des dimensions, on indique son nombre de composants :

```
t = new Tc [N1][N2] ... [Nn];
```

La déclaration de la matrice de réels à `m` lignes et `n` colonnes de la section 11.1 s'écrit en JAVA comme suit :

```
double [][] matrice = new double [m][n];
```

L'accès aux éléments de la matrice se fait par une double indexation, dénotée `matrice[i][j]`, où `i` et `j` sont deux indices définis, respectivement, sur les intervalles `[0,m-1]` et `[0,n-1]`.

Notez que le nom de la variable qui désigne le tableau (*e.g.* `matrice`), ou les composants d'un tableau à plusieurs dimensions (*e.g.* `matrice[1]`) sont des *références* sur des tableaux à une dimensions. Le modèle n'est donc pas tout à fait celui de tableaux de tableaux. En revanche, cela autorise un nombre de composants différents par dimension. Il n'est pas obligatoire de créer toutes les composantes en une seule fois. Il est ainsi possible d'écrire :

```
double [][] matrice = new double [m][];
```

La variable `matrice` désigne un tableau de `m` composants initialisés à **null**. Par la suite, chacun des composants pourra être créé individuellement.

```
matrice[0] = new double[5];
 ...
matrice[3] = new double[10];
```

La première ligne de la matrice possède 5 colonnes, alors que la quatrième en possède 10.

11.6 EXEMPLES

11.6.1 Initialisation d'une matrice

Soit la déclaration de la matrice à `m` lignes et `n` colonnes suivante :

```
variable
   matrice type tableau [[1,m],[1,n]] de réel
```

On désire initialiser tous les éléments de la matrice à une valeur réelle v. Pour cela, il est nécessaire de parcourir toutes les lignes, et pour chaque ligne toutes les colonnes, à l'aide de deux énoncés itératifs **pourtout** emboîtés.

```
Algorithme init matrice
   {initialisation à la valeur v de tous les éléments de la matrice}
   pourtout i de 1 à m faire
       pourtout j de 1 à n faire
           {∀x∈[1,i-1], ∀y∈[1,j-1], matrice[x,y]=v}
           matrice[i,j] ← v
       finpour
   finpour
   {∀x∈[1,m], ∀y∈[1,n], matrice[x,y]=v}
```

Le fragment de code JAVA correspondant à l'algorithme précédent est :

```
// initialisation à la valeur v de tous les éléments de la matrice
for (int i = 0; i<m; i++)
    for (int j = 0; j<n; j++)
       // ∀x∈[0,i-1], ∀y∈[0,j-1], matrice[x,y]=v
       matrice[i][j]=v;
// ∀x∈[0,m-1], ∀y∈[0,n-1], matrice[x,y]=0
```

11.6.2 Matrice symétrique

On désire tester si une matrice carrée est symétrique ou non par rapport à la diagonale principale. On rappelle qu'une matrice carrée $m(n,n)$ est symétrique si $\forall i,j \in [1,n], m_{ij} = m_{ji}$.

L'algorithme consiste à parcourir, à l'aide de deux boucles emboîtées, la demi-matrice supérieure (ou inférieure) et vérifier que $m[i,j] = m[j,i]$. Notez qu'il est inutile de tester les éléments de la diagonale (*i.e.* $i = j$). D'autre part, le nombre d'itérations n'étant pas connu à l'avance, l'énoncé **pourtout** n'est pas adapté au parcours des lignes et des colonnes. La complexité de cet algorithme est $\mathcal{O}(n^2)$.

```
Algorithme symétrique
   {teste si une matrice est symétrique ou non}
   variables l, c type [1,n]
             passymétrique type booléen
   l ← 1
   passymétrique ← faux
   répéter
         l ← l+1
         c ← 1
         répéter
               si matrice[l,c] ≠ matrice[c,l] alors
                     passymétrique ← vrai
               sinon
                     c ← c+1
```

```
                finsi
            jusqu'à c = l ou passymétrique
              {passymétrique ou ∀i,j∈[1,l], matrice[i,j]=matrice[j,i]}
        jusqu'à l = n ou passymétrique
        {passymétrique ou ∀i,j∈[1,n], matrice[i,j]=matrice[j,i]}
        rendre non passymétrique
```

On donne en JAVA la programmation de la fonction `symétrique` qui teste une matrice passée en paramètre. Notez la suppression de la variable booléenne `passymétrique`.

```
// teste si une matrice est symétrique ou non
public boolean symétrique(int [][] matrice) {
    int l = 0;
    do {
        l++;
        int c = 0;
        do {
            if (matrice[l][c] != matrice[c][l])
                // la matrice n'est pas symétrique
                return false;
            c++;
        } while (c < l);
        // ∀i,j∈[0,l], matrice[i,j]=matrice[j,i]
    } while (l < matrice.length-1);
    // ∀i,j∈[0, matrice.length-1], matrice[i,j]=matrice[j,i]
    // la matrice est symétrique
    return true;
}
```

11.6.3 Produit de matrices

Soient trois matrices $a(m,p)$, $b(p,n)$ et $c(m,n)$, on désire programmer le produit $c = a \times b$. Rappelons que les éléments de la matrice c sont tels que

$$\forall i \in [1,m], \forall j \in [1,n], c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

L'algorithme suit précisément cette équation, qui sera l'invariant de boucle. Il possède trois énoncés **pourtout** emboîtées et sa complexité est $\mathcal{O}(n^3)$.

```
constantes
   m = ? {nombre lignes de la matrice a}
   n = ? {nombre lignes de la matrice b et
          nombre colonnes de la matrice a}
   p = ? {nombre colonnes de la matrice c}
variables
  a type tableau [[1,m],[1,p]] de réel
  b type tableau [[1,p],[1,n]] de réel
  c type tableau [[1,m],[1,n]] de réel
  somme type réel
```

```
pourtout i de 1 à m faire
    pourtout j de 1 à n faire
        {∀x∈[1,i-1], ∀y∈[1,j-1], c[x,y]=∑_{k=1}^{p} a[x,k]×b[k,y]}
        somme ← 0
        pourtout k de 1 à p faire
            somme ← somme+a[i,k]×b[k,j]
        finpour
        c[i,j] ← somme
    finpour
finpour
{∀i∈[1,m], ∀j∈[1,n], c[i,j]=∑_{k=1}^{p} a[i,k]×b[k,j]}
```

Notez qu'il existe une autre méthode, celle de STRASSEN, basée sur la décomposition de la matrice en quatre quadrants, de complexité inférieure à $\mathcal{O}(n^3)$.

11.6.4 Carré magique

On appelle *carré magique*[2] une matrice carrée d'ordre n, contenant les entiers compris entre 1 et n^2, telle que les sommes des entiers de chaque ligne, de chaque colonne et des deux diagonales sont identiques. La matrice suivante est un carré magique d'ordre 3 :

4	9	2
3	5	7
8	1	6

Nous présentons une méthode de complexité $\mathcal{O}(n^2)$ pour créer des carrés magiques d'ordre impair. Le chiffre 1 est mis dans la case située une ligne en dessous de la case centrale. Lorsqu'on a placé un entier x dans une case de coordonnées (i,j), on place $x+1$ dans la case $(l,k) = (i+1,j+1)$. Cependant, ces coordonnées ne sont pas nécessairement valides. Si un indice est égal à la valeur $n+1$, on lui affecte la valeur 1 ; et si la case est déjà occupée, on essaie alors de placer le nombre en $(l+1,k-1)$, mais si $k-1=0$ alors k prend la valeur n. Écrivons formellement l'algorithme selon cette méthode. Une case libre a pour valeur 0.

Algorithme Carré Magique

```
{on considère le carré initialisé à 0}
{écrire le premier nombre dans la première case}
j ← (n+1)/2
i ← j+1
carré[i,j] ← 1
{placer les nombres suivants de 2 à n²}
pourtout k de 2 à n²
    {calculer les prochaines coordonnées (i+1,j+1)}
    i ← i+1
```

2. Les carrés magiques sont très anciens, puisqu'on trouve leur trace, il y a près de 3 000 ans, sur la carapace d'une tortue dans la légende chinoise de LO SHU. En Europe, le premier carré magique apparaît en 1514 sur une gravure du peintre allemand A. DÜRER. Si durant de nombreux siècles ces carrés étaient attachés à des superstitions divines, à partir du XVII[e] siècle, ils ont fait l'objet de nombreuses études mathématiques.

```
        si i>n alors i ← 1 finsi
        j ← j+1
        si j>n alors j ← 1 finsi
        {est-ce que la place carré[i,j] est occupée ?}
        {si oui, trouver une place libre}
        si carré[i,j]≠0 alors
           i ← i+1
           si i>n alors i ← 1 finsi
           j ← j-1
           si j=0 alors j ← n finsi
        finsi
        {carré[i,j] est la place de l'entier k}
        carré[i,j] ← k
    finpour
```

La programmation en JAVA de cet algorithme consistera à définir une classe `CarréMagique` avec comme attribut une matrice d'entiers, et un constructeur qui crée le carré magique. Cette classe, complétée par la méthode `toString` est entièrement donnée ci-dessous :

```
public class CarréMagique {
    private int [][] carré;
    public CarréMagique(int n) {
        // Rôle : crée un carré magique d'ordre n
        carré = new int[n][n];
        int j=n/2, i=j+1;
        // on place le premier nombre dans la première case
        carré[i][j]=1;
        // puis les suivants de 2 à n²
        final int nAuCarré=n*n;
        for (int k=2; k<=nAuCarré; k++) {
            // est-ce que les nouvelles coordonnées i et j sont < n?
            if (++i==n) i=0;
            if (++j==n) j=0;
            // est-ce que la place est occupée?
            // si oui, trouver une place libre
            if (carré[i][j]!=0) {
                if (++i==n) i=0;
                if (--j<0)  j=n-1;
            }
            // carré[i][j] est la place de l'entier k
            carré[i][j]=k;
        }
    }
    public String toString() {
        String s="";
        for (int i=0; i<carré.length; i++) {
            for (int j=0; j<carré.length; j++)
                s += carré[i][j] + "␣";
```

```
                s += '\n';
            }
            return s;
        }
} // fin classe CarréMagique
```

11.7 EXERCICES

Exercice 11.1. Donnez en fonction de l'ordre n d'un carré magique la valeur de la somme des cases d'une ligne (ou colonne ou diagonale).

Exercice 11.2. On définit en JAVA la classe `Matrice` possédant trois attributs : un tableau à deux dimensions qui contiendra les éléments de la matrice, son nombre de lignes et de colonnes :

```
class Matrice {
    // Invariant: this est une matrice (lignes,colonnes)
    private double [][] m;
    public int nbLignes, nbColonnes;
} // fin classe Matrice
```

Remarquez que la représentation de la matrice est privée, et inconnue des utilisateurs de cette classe. Quel est l'intérêt de masquer aux clients la représentation des éléments de la matrice ? Si demain, la représentation change, le code du client ne changera pas et restera valide. À la place d'un tableau à deux dimensions, on pourrait imaginer une autre représentation des éléments de la matrice, en particulier si elle est creuse[3].

Si nous voulons préserver cette indépendance vis-à-vis d'une représentation particulière des données, nous devons définir des méthodes qui la maintiennent.

Écrivez les méthodes `prendre` et `mettre`. La première retourne la valeur de l'élément (i,j), la seconde lui affecte une valeur donnée.

Exercice 11.3. En utilisant les méthodes précédentes, écrivez les méthodes `symétrique` et `produit` dont les algorithmes sont décrits plus haut. La première méthode teste la symétrie de la matrice courante, et la seconde retourne une matrice (*i.e.* de type `Matrice`) produit de la matrice courante et d'une seconde passée en paramètre. Pour que le produit de deux matrices soit valide, vous vérifierez si le nombre de colonnes de la première est égal au nombre de lignes de la seconde.

Exercice 11.4. Écrivez une méthode qui retourne le vecteur résultat du produit de la matrice courante par un vecteur passé en paramètre. On rappelle que le produit d'une matrice a de dimension $m \times n$ et du vecteur v de dimension n est le vecteur v' de dimension m dont les composantes sont définies par :

$$v'_i = \sum_{k=1}^{n} a_{ik} \times v_k$$

3. Une matrice creuse est une matrice dont la majorité des éléments sont égaux à zéro.

Exercice 11.5. Soit le système linéaire de n équations à n inconnues suivant :

$$\begin{array}{cccc} a_{11}x_1 & a_{12}x_2 & \dots & a_{1n}x_n = b_1 \\ a_{21}x_1 & a_{22}x_2 & \dots & a_{2n}x_n = b_2 \\ \vdots & \vdots & & \vdots \\ a_{n1}x_1 & a_{n2}x_2 & \dots & a_{nn}x_n = b_n \end{array}$$

On représente ce système par l'équation $AX = B$, où A est la matrice des coefficients a_{ij} et B est le vecteur des coefficients b_i.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

La résolution de ce système linéaire par la méthode de GAUSS[4] se fait en deux étapes. La première transforme la matrice du système en une matrice triangulaire avec seulement des uns sur sa diagonale. La seconde calcule par substitution les solutions du système, à partir de la matrice triangularisée, en partant de la dernière équation jusqu'à la première. Ainsi, après triangularisation, le système est transformé en un système équivalent :

$$\begin{pmatrix} 1 & a'_{12} & \dots & a'_{1n} \\ 0 & 1 & \dots & a'_{2n} \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{pmatrix}$$

Pour trianguliariser la matrice A, on procède de façon itérative du rang 1 au rang n. À la k^{e} étape, la sous-matrice $(k-1,k-1)$ est triangularisée et les opérations à effectuer pour poursuivre la triangularisation sont les suivantes :

- choisir le pivot ;
- normaliser la ligne, c'est-à-dire diviser la ligne k du système par le pivot (*i.e.* les a_{kj} et b_k) ;
- pour toute ligne i du système allant de $k+1$ à n, lui soustraire la ligne de rang k multipliée par a_{ik}.

Notez que ces opérations ne modifient évidemment pas la solution du système d'équations. Celle-ci se calcule par substitution « en remontant » à partir de la dernière équation : $x_n = b'_n$, $x_{n-1} = b'_{n-1} - a'_{n-1,n}x_n$, *etc.* L'algorithme de C. F. GAUSS a la forme suivante :

Algorithme `GaussRSL(données A, B résultat X)`

```
{Antécédent : A, B données du système linéaire}
{Conséquent : X solution du système AX=B}
```

4. C. F. GAUSS, astronome, mathématicien et physicien allemand (1777-1855).

```
{triangularisation de la matrice A}
pourtout k de 1 à n faire
   pivot ← A[k,k] { A[k,k] est le pivot}
   {normaliser la ligne k de la matrice A et}
   {du vecteur B par le pivot tel que A[k,k]=1}
   ...
   pourtout i ← k+1 à n faire
   {soustraire à la ie ligne de la matrice A et}
   {du vecteur B la ligne k multipliée par A[i,k]}
   ...
   finpour
finpour

{calcul de la solution X}
pourtout k de n à 1 faire
   sol ← B[k]
   {calcul de la solution sol par substitution en remontant}
   {jusqu'à la k+1e ligne}
   ...
   X[k] ← sol
finpour
```

Complétez l'algorithme précédent, avec dans un premier temps, a_{kk} comme valeur de pivot.

Que se passe-t-il si le pivot a_{kk} est nul ? Une solution est de chercher un pivot $a_{vk} \neq 0$ avec $k+1 \leqslant v \leqslant n$, puis, d'échanger la ligne v qui contient ce pivot avec la ligne k. Notez que si on ne peut trouver de pivot différent de zéro, le système est *lié* et n'admet pas une solution unique. D'une façon générale, pour diminuer les risques d'erreurs dans les calculs numériques, on choisit le pivot le plus grand en valeur absolue entre les lignes k et n. Modifiez l'algorithme en conséquence. Quelle est la complexité de cet algorithme ?

Exercice 11.6. On désire calculer l'inverse d'une matrice A. Pour cela, on procède de la même manière que dans l'exercice précédent, mais B est la matrice identité. Pour inverser la matrice A, la méthode de GAUSS doit se poursuivre pour obtenir une double triangularisation (inférieure et supérieure) de A. Le système initial est le suivant :

$$\left(\begin{array}{cccc|cccc} a_{11} & a_{12} & \dots & a_{1n} & 1 & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & a_{2n} & 0 & 1 & \dots & 0 \\ \vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots \\ a_{n1} & a_{12} & \dots & a_{nn} & 0 & 0 & \dots & 1 \end{array}\right)$$

La double triangularisation produit le système suivant, où A est transformé en matrice identité, et B en la matrice inverse recherchée :

$$\left(\begin{array}{cccc|cccc} 1 & 0 & \dots & 0 & b_{11} & b_{12} & \dots & b_{1n} \\ 0 & 1 & \dots & 0 & b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & 1 & b_{n1} & b_{n2} & \dots & b_{nn} \end{array}\right)$$

Écrivez l'algorithme d'inversion d'une matrice.

Exercice 11.7. Appliquez la méthode de la double triangularisation pour résoudre un système d'équations linéaires. Comparez la complexité de cet algorithme avec celui de l'exercice 11.4.

Chapitre **12**

Héritage

Une des principales qualités constitutives des langages à objets est la *réutilisabilité*, c'est-à-dire la réutilisation de classes existantes. La réutilisation de composants logiciels déjà programmés et fiables permet une économie de temps et d'erreurs dans la construction de nouveaux programmes. Chaque année, des millions de lignes de code sont écrites et seul un faible pourcentage est original. Si cela se conçoit dans un cadre pédagogique, ça l'est beaucoup moins dans un environnement industriel. Des algorithmes classiques sont reprogrammés des milliers de fois, avec les risques d'erreur que cela comporte, au lieu de faire partie de bibliothèques afin d'être mis à la disposition de programmeurs. La notion d'héritage, que nous allons aborder dans ce chapitre, est l'outil qui facilitera la conception de programmes par réutilisation.

12.1 CLASSES HÉRITIÈRES

Dans le chapitre 7, nous avons défini une classe pour représenter et manipuler des rectangles. Si, maintenant, nous désirons représenter des carrés, il nous faut définir une nouvelle classe. Chaque carré est caractérisé par la longueur de son côté, son périmètre, sa surface, *etc.* La classe pour représenter les carrés est définie comme suit :

```
classe Carré
   {Invariant de classe : côté ⩾ 0}
   public périmètre, surface, côté

   {l'attribut}
   côté type réel
```

```
    {le constructeur}
    constructeur Carré(donnée c : réel)
        côté ← c
    fincons

    {les méthodes}
    fonction périmètre() : réel
    {Rôle: retourne le périmètre du carré}
        rendre 4×côté
    finfonc {périmètre}

    fonction surface() : réel
    {Rôle: retourne la surface du carré}
        rendre côté×côté
    finfonc {surface}
finclasse Carré
```

Vous pouvez constater que cette classe ressemble fortement à la classe `Rectangle`. Ceci est normal dans la mesure où un carré est un rectangle particulier dont la largeur est égale à la longueur. Aussi, plutôt que de définir entièrement une nouvelle classe, il est légitime de *réutiliser* certaines des caractéristiques d'un rectangle pour définir un carré. L'*héritage* est une relation entre deux classes qui permet à une classe de réutiliser les caractéristiques d'une autre. Nous définirons la classe `Carré` comme suit :

```
classe Carré hérite de Rectangle
    {Invariant de classe : longueur = largeur ⩾ 0}
    {le constructeur}
    constructeur Carré(donnée c : réel)
          Rectangle(c,c)
    fincons
finclasse Carré
```

Tous les attributs et toutes les méthodes de la classe `Rectangle`, la classe *parent*, sont accessibles depuis la classe `Carré`, le *descendant*. On dit que la classe `Carré` *hérite*[1] de la classe `Rectangle`. La classe `Carré` ne définit que son constructeur, qui appelle celui de sa classe parent, et hérite automatiquement des attributs, `largeur` et `longueur`, ainsi que des méthodes `périmètre` et `surface`.

La figure 12.1 montre de façon graphique la relation d'héritage entre les classes `Rectangle` et `Carré`. Chaque classe est représentée par une boîte qui porte son nom et l'orientation de la flèche signifie *hérite de*.

La déclaration d'un carré `c` et le calcul de sa surface sont alors obtenus par :

```
variable c type Carré créer Carré(5)

... c.surface() ...
```

La réutilisabilité des classes déjà fabriquées est un intérêt majeur de l'héritage. Il évite une perte de temps dans la réécriture de code déjà existant, et évite ainsi l'introduction de

1. On dit également *dérive* et on parle alors de classe dérivée.

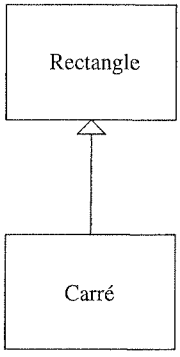


Figure 12.1 – *Relation d'héritage entre les classes* `Rectangle` *et* `Carré`.

nouvelles erreurs dans les programmes. L'héritage peut être vu comme un procédé de *factorisation*, par la mise en commun des caractéristiques communes des classes.

Comme pour une généalogie humaine, une classe héritière peut avoir ses propres descendants, créant ainsi un véritable arbre généalogique. La relation d'héritage est une relation transitive : si une classe B hérite d'une classe A, et si une classe C hérite de la classe B, alors la classe C hérite également de A. Dans beaucoup de langages de classe, toutes les classes possèdent un ancêtre commun, une classe souvent appelée *Object*, qui est la racine de l'arborescence d'héritage (voir la figure 12.2).

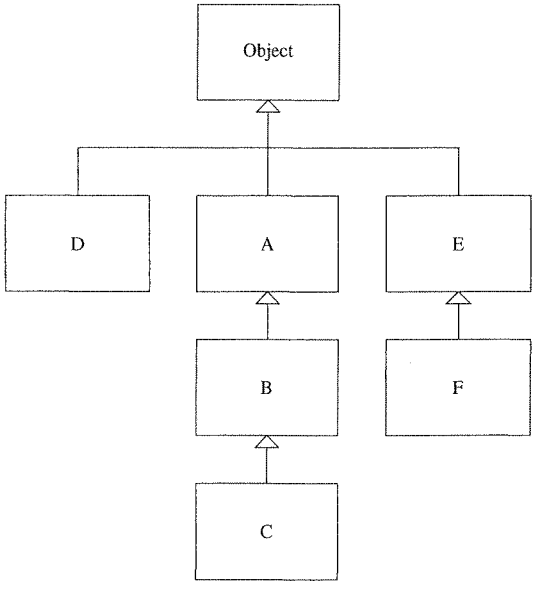


Figure 12.2 – *Un ancêtre commun : la classe Object.*

Évidemment, les classes héritières peuvent définir leurs propres caractéristiques. La classe `Carré` possède déjà son propre constructeur, mais pourra définir, par exemple, une méthode de mise à jour du côté d'un carré.

```
classe Carré hérite de Rectangle
   {Invariant de classe : longueur = largeur ⩾ 0}
   public changerCôté
   {le constructeur}
   constructeur Carré(donnée c : réel)
         Rectangle(c,c)
   fincons

   procédure changerCôté(donnée c : réel)
   {Rôle: met la valeur du côté à la valeur c}
      changerLargeur(c)
      changerLongueur(c)
   finproc
finclasse Carré

...
c.changerCôté(10)
```

La relation d'héritage peut donc aussi être considérée comme un mécanisme d'*extension* de classes existantes, mais également de *spécialisation*. Les informations les plus générales sont mises en commun dans des classes parentes. Les classes se spécialisent par l'ajout de nouvelles fonctionnalités. Il est important de comprendre que n'importe quelle classe ne peut étendre ou spécialiser n'importe quelle autre classe. La classe `Carré` qui hériterait d'une classe `Individu` n'aurait aucun sens. Le mécanisme d'héritage permet de conserver une cohérence entre les classes ainsi mises en relation.

12.2 REDÉFINITION DE MÉTHODES

Lorsqu'une classe héritière désire modifier l'implémentation d'une méthode d'une classe parent, il lui suffit de redéfinir cette méthode. La *redéfinition* d'une méthode est nécessaire si on désire adapter son action à des besoins spécifiques. Imaginons, par exemple, que la classe `Rectangle` possède une méthode d'affichage, la classe `Carré` peut redéfinir cette méthode pour l'adapter à ses besoins.

```
classe Rectangle
   ...
   procédure afficher()
   {Rôle: affiche une description du rectangle courant}
       écrire("rectangle de largeur", largeur,
                      "et de longueur" , longueur)
   finproc
  ...
finclasse Rectangle
```

```
classe Carré hérite de Rectangle
    ...
    procédure afficher()
    {Rôle: affiche une description du carré courant}
        écrire("carré de côté égal à" , largeur)
    finproc
    ...
finclasse Carré
```

Dans le fragment de code suivant :

```
variable c type Carré créer Carré(5)
variable r type Rectangle créer Rectangle(2,4)

r.afficher()
c.afficher()
```

il est clair que c'est la méthode `afficher` de la classe `Rectangle` qui s'applique à l'objet `r` et celle de la classe `Carré` qui s'applique à l'objet `c`. Notez que les redéfinitions permettent de changer la mise en œuvre des actions, tout en préservant leur sémantique. Ainsi, la méthode `afficher` de la classe `Carré` ne devra pas calculer, par exemple, la surface d'un carré.

12.3 RECHERCHE D'UN ATTRIBUT OU D'UNE MÉTHODE

Si la méthode que l'on désire appliquer à une occurrence d'un objet d'une classe C n'est pas présente dans la classe, celle-ci devra appartenir à l'un de ses ancêtres.

D'une façon générale, chaque fois que l'on désire accéder à un attribut ou une méthode d'une occurrence d'objet d'une classe C, il (ou elle) devra être défini(e), soit dans la classe C, soit dans l'un de ses ancêtres. Si l'attribut ou la méthode n'appartient pas aux classes parentes, l'attribut ou la méthode n'est pas trouvé et c'est une erreur de programmation.

S'il y a eu des redéfinitions de la méthode, sa première apparition en remontant l'arborescence d'héritage est celle qui sera choisie. Ainsi, avec les déclarations suivantes :

```
variable c type Carré
variable r type Rectangle
```

`c.périmètre()`, provoque l'exécution de la méthode `périmètre` définie dans la classe `Rectangle`, alors que `r.changerCôté()` provoque une erreur.

12.4 POLYMORPHISME ET LIAISON DYNAMIQUE

Dans certains langages de programmation, une variable peut désigner, à tout moment au cours de l'exécution d'un programme, des valeurs de n'importe quel type. De tels langages sont dits *non typés* ou encore *polymorphiques*[2]. En revanche, les langages dans lesquels une

2. Du grec *poly* plusieurs et *morphe* forme.

variable ne peut désigner qu'un seul type de valeur sont dits *typés* ou *monomorphiques*. Ces derniers offrent plus de sécurité dans la construction des programmes dans la mesure où les vérifications de cohérence de type sont faites dès la compilation, alors qu'il faut attendre l'exécution du programme pour les premiers.

Dans un langage de classe typé, comme par exemple JAVA, le *polymorphisme* est contrôlé par l'héritage. Ainsi, une variable de type `Rectangle` désigne bien évidemment des occurrences d'objets de type `Rectangle`, mais pourra également désigner des objets de type `Carré`. Si la variable `r` est de type `Rectangle`, et la variable `c` de type `Carré`, l'affectation `r←c` est valide. Cette affectation se justifie puisqu'un carré est en fait un rectangle dont la largeur et la longueur sont égales. La relation d'héritage qui lie les rectangles et les carrés est vue comme une relation *est-un*. Un carré *est un* rectangle (spécialisé). En revanche, l'affectation inverse, `c←r`, n'est pas licite, puisqu'un rectangle n'est pas (nécessairement) un carré.

Le polymorphisme des langages de classe typés permet d'assouplir le système de type, tout en conservant un contrôle de type rigoureux. Imaginons que l'on veuille manipuler des formes géométriques diverses. Nous définirons la classe `Forme` pour représenter des formes géométriques quelconques. La classe `Rectangle` héritera de cette classe, puisqu'un rectangle est bien une forme géométrique. De même, nous définirons des classes pour représenter des ellipses et des cercles. L'arbre d'héritage de ces classes est donné par la figure 12.3.

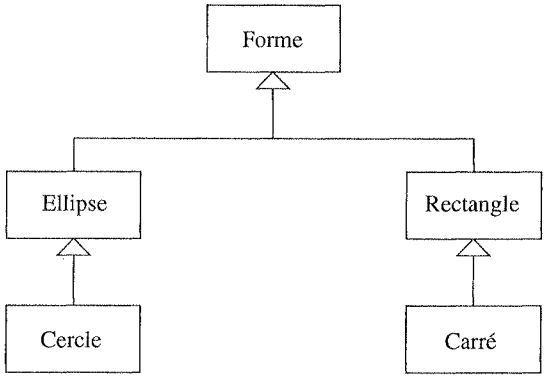


Figure 12.3 – *Arbre d'héritage des figures géométriques.*

Les éléments d'un tableau de type `Forme` pourront désigner, à tout moment, des ellipses, des cercles, des rectangles ou encore des carrés. Sans le polymorphisme, il aurait fallu déclarer autant de tableaux qu'il existe de formes géométriques.

```
t type tableau[ [1,max] ] de Forme

t[1] ← créer Rectangle(3,10)
t[2] ← créer Cercle(8)
...
t[1] ← t[2] {t[1] désigne (peut-être) un cercle}
```

Chacune des classes, qui hérite de `Forme`, est pourvue d'une méthode `afficher` qui produit un affichage spécifique de la forme qu'elle définit. S'il est clair qu'après la première affectation :

```
t[1] ← créer Rectangle(3,10)
```

l'instruction `t[1].afficher()` produit l'affichage d'un rectangle, il n'en va pas de même si cette même instruction est exécutée après la dernière affectation :

```
t[1] ← t[2]
```

Pour cette dernière, la méthode à appliquer ne peut être connue qu'à l'exécution du programme, au moment où l'on connaît la nature de l'objet qui a été affecté à `t[1]`, c'est-à-dire l'objet qu'il désigne réellement. Ce sera la méthode `afficher` de la classe `Cercle`, si `t[2]` désigne bien un cercle au moment de l'affectation. Lorsqu'il y a des redéfinitions de méthodes, c'est au moment de l'exécution que l'on connaît la méthode à appliquer. Elle est déterminée à partir de la forme dynamique de l'objet sur lequel elle s'applique. Ce mécanisme, appelé *liaison dynamique*, s'applique à des méthodes qui possèdent exactement les mêmes signatures, proposant dans des classes *différentes* d'une même ascendance, la mise en œuvre d'une même opération. Il a pour intérêt majeur une *utilisation* des méthodes redéfinies *indépendamment* des objets qui les définissent. On voit bien dans l'exemple précédent, qu'il est possible d'afficher ou de calculer le périmètre d'une forme sans se soucier de sa nature exacte.

12.5 CLASSES ABSTRAITES

Dans la section précédente, nous n'avons pas rédigé le corps des méthodes de la classe `Forme`. Puisque cette classe représente des formes quelconques, il est bien difficile d'écrire les méthodes `surface` ou `afficher`. Toutefois, ces méthodes doivent être nécessairement définies dans cette classe pour qu'il y ait polymorphisme ; la variable `t` est un tableau de `Forme` et `t[1].afficher()` doit être définie. Il est toujours possible de définir le corps des méthodes vide, mais alors rien ne garantit que les méthodes seront effectivement redéfinies par les classes héritières.

Une classe *abstraite* est une classe très générale qui décrit des propriétés qui ne seront définies que par des classes héritières, soit parce qu'elle ne sait pas comment le faire (*e.g.* la classe `Forme`), soit parce qu'elle désire proposer différentes mises en œuvres (voir les types abstraits, chapitre 16). Nous définirons, par exemple, la classe `Forme` comme suit :

```
classe abstraite Forme
    public périmètre, surface, afficher
    {les méthodes abstraites}
    fonction périmètre() : réel
    fonction surface() : réel
    procédure afficher()
finclasse abstraite Forme
```

Les méthodes d'une telle classe sont appelées *méthodes abstraites*, et seuls les en-têtes sont spécifiés. Une classe abstraite ne peut être instanciée, il n'est donc pas possible de créer

des objets de type `Forme`. De plus, les classes héritières (*e.g.* `Rectangle`) sont dans l'obligation de redéfinir les méthodes de la classe abstraite, sinon elles seront considérées elles-mêmes comme abstraites, et ne pourront donc pas être instanciées.

12.6 HÉRITAGE SIMPLE ET MULTIPLE

Il arrive fréquemment qu'une classe doive posséder les caractéristiques de plusieurs classes parentes distinctes. Une figure géométrique formée d'un carré avec en son centre un cercle d'un rayon égal à celui du côté du carré, pourrait être décrite par une classe qui hériterait à la fois des propriétés des carrés et des cercles (voir la figure 12.4). Cette classe serait par exemple contrainte par la relation *largeur = longueur = diamètre*.

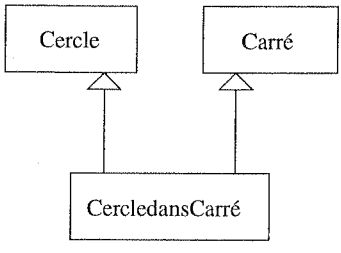


Figure 12.4 – *Graphe d'héritage CercledansCarré.*

Lorsqu'une classe ne possède qu'une seule classe parente, l'héritage est *simple*. En revanche, si une classe peut hériter de plusieurs classes parentes différentes, l'héritage est alors *multiple*. Avec l'héritage multiple, les relations d'héritage entre les classes ne définissent plus une simple arborescence, mais de façon plus générale un graphe, appelé *graphe d'héritage*[3].

L'héritage multiple introduit une complexité non négligeable dans le choix de la méthode à appliquer en cas de conflit de noms ou d'héritage répété. Pour le programmeur, le choix d'une méthode a appliquer peut ne pas être évident. C'est pour cela que certains langages de programmation, comme JAVA[4], ne le permettent pas.

12.7 HÉRITAGE ET ASSERTIONS

Le mécanisme d'héritage introduit de nouvelles règles pour la définition des assertions des classes héritières et des méthodes qu'elles comportent.

3. Voir les chapitres 18 et 19 qui décrivent les types abstraits graphe et arbre.

4. Toutefois, ce langage définit la notion d'*interface* qui permet de mettre en œuvre une forme particulière de l'héritage multiple.

12.7.1 Assertions sur les classes héritières

L'invariant d'une classe héritière est la conjonction des invariants de ses classes parentes et de son propre invariant. Dans notre exemple, l'invariant de la classe `Carré` est celui de la classe `Rectangle`, *i.e.* la largeur et la longueur d'un rectangle doivent être positives ou nulles, *et* de son propre invariant, *i.e.* ces deux longueurs doivent être égales.

12.7.2 Assertions sur les méthodes

Les règles de définition des antécédents et des conséquents sur les méthodes doivent être complétées dans le cas particulier de la redéfinition. Nous prendrons ici les règles données par B. MEYER [Mey97].

Une assertion A est plus forte qu'une assertion B, si A implique B. Inversement, nous dirons que B est plus faible. Lors d'une redéfinition d'une méthode m, que nous appellerons m', il faudra que :

(1) l'antécédent de m' soit plus faible ou égal que celui de m ;

(2) le conséquent de m' soit plus fort ou égal que celui de m.

Pour comprendre cette règle, il faut la voir à la lumière de la liaison dynamique. Une variable déclarée de type classe A peut appeler la méthode m, mais exécuter sa redéfinition m' dans la classe héritière B sous l'effet de la liaison dynamique. Cela indique que toute assertion qui s'applique à m *doit* également s'appliquer à m'. Aussi, la règle (1) indique que m' doit accepter l'antécédent de m, et la règle (2) que m' doit également vérifier le conséquent de m.

12.8 RELATION D'HÉRITAGE OU DE CLIENTÈLE

Lors de la construction d'un programme, comment choisir les relations à établir entre les classes ? Une classe A doit-elle hériter d'une classe B, ou en être la cliente ? Une première réponse est de dire que si on peut appliquer la relation *est-un*, sans doute faudra-t-il utiliser l'héritage. Dans notre exemple, un carré *est-un* rectangle particulier dont la largeur et la longueur sont égales. La classe `Carré` hérite de la classe `Rectangle`. En revanche, si c'est une relation *a-un* qui doit s'appliquer, il faudra alors établir une relation de clientèle. Une voiture *a-un* volant, une école *a-des* élèves. La classe `Voiture`, qui représente des automobiles, possédera un attribut `volant` qui le décrit. De même, les élèves d'une école peuvent être représentés par la classe `Élèves`, et la classe `École` possédera un attribut pour désigner tous les élèves de l'école.

Cette règle possède l'avantage d'être simple et nous l'utiliserons chaque fois que cela est possible. Toutefois, elle ne pourra être appliquée systématiquement, et nous verrons par la suite des cas où elle devra être mise en défaut.

12.9 L'HÉRITAGE EN JAVA

En JAVA, l'héritage est simple et toutes les classes possèdent *implicitement* un ancêtre commun, la classe `Object`. On retrouve dans ce langage les concepts exposés précédemment, et dans cette section, nous n'évoquerons que ses spécificités.

Les classes héritières, ou *sous-classes*, comportent dans leur en-tête le mot-clé **`extends`**[5] suivi du nom de la classe parente. Le constructeur d'une sous-classe peut faire appel à un constructeur de sa classe parente, la *super-classe* appelée **`super`**. S'il ne le fait pas, un appel implicite au constructeur par défaut de la classe parente, c'est-à-dire **`super()`**, aura systématiquement lieu. Notez que le constructeur par défaut de la classe mère doit alors exister. La classe `Carré` s'écrit en JAVA :

```
public class Carré extends Rectangle {
    /** Invariant de classe : longueur = largeur ⩾ 0 */
    // le constructeur
    public Carré(double c) {
         // appel du constructeur de Rectangle
         super(c,c);
    }
    public void changerCôté(double côté)
    // Rôle: met à jour le côté du carré courant
    {
        changerLargeur(côté);
        changerLongueur(côté);
    }
    public String toString()
    // Rôle: convertit le carré courant en chaîne de caractères
    {
        return "carré␣de␣côté␣égal␣à" + largeur;
    {
} // fin classe Carré
```

La création d'un carré `c` de côté 7 et l'affichage de sa surface s'écriront comme suit :

```
Carré c = new Carré(7);
System.out.println(c.surface());
```

La redéfinition des méthodes dans les sous-classes ne peut se faire qu'avec des méthodes qui possèdent exactement les mêmes signatures. La liaison dynamique est donc mise en œuvre sur des méthodes qui possèdent les mêmes en-têtes et qui diffèrent par leurs instructions, comme par exemple la méthode `toString` des classes `Rectangle` et `Carré`.

Les classes abstraites sont introduites par le mot-clé **`abstract`**, de même que les méthodes. Notez que seule une partie des méthodes peut être déclarée abstraite, la classe demeurant toutefois abstraite. La classe `Forme` possède la déclaration suivante :

```
abstract class Forme {
    public abstract double périmètre();
    public abstract double surface();
}
```

5. Ce qui indique bien l'idée d'extension de classe.

Remarquez l'absence de la méthode `toString`, puisque celle-ci est héritée de la classe `Object`.

Contrairement aux classes abstraites, les *interfaces* sont des classes dont *toutes* les méthodes sont *implicitement* abstraites et qui ne peuvent posséder d'attributs, à l'exception de constantes. Les interfaces permettent, d'une part, une forme simplifiée de l'héritage multiple, et d'autre part la *généricité*. Nous reparlerons de ces deux notions plus loin, à partir du chapitre 16.

L'interface de programmation d'application de JAVA (API[6]) est une hiérarchie de classes qui offrent aux programmeurs des classes préfabriquées pour manipuler les fichiers, construire des interfaces graphiques, établir des communications réseaux, *etc.*

La classe `Object` est au sommet de cette hiérarchie. Elle possède en particulier deux méthodes `clone()` et `equals(Object o)`. La première permet de dupliquer l'objet courant, et la seconde de comparer si l'objet passé en paramètre est égal à l'objet courant. Ces méthodes sont nécessaires puisque les opérations d'affectation et d'égalité mettant en jeu deux opérandes de type classe manipulent des références et non pas les objets eux-mêmes.

Il n'est pas question de présenter ces classes ici ; ce n'est d'ailleurs pas l'objet de cet ouvrage. Toutefois, il faut mentionner que les types simples primitifs du langage possèdent leur équivalent objet dont la correspondance est donnée par la table 12.1.

Type primitif	*Classe correspondante*
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

TAB. 12.1 – *Correspondance types primitifs – classes.*

Règles de visibilité

Nous avons déjà vu qu'un membre d'une classe pouvait être qualifié **`public`**, pour le rendre visible par n'importe quelle classe, et **`private`**, pour restreindre sa visibilité à sa classe de définition. Le langage JAVA propose une troisième qualification, **`protected`**, qui rend visible le membre par toutes les classes héritières de sa classe de définition.

En fait, les membres qualifiés **`protected`** sont visibles par les héritiers de la classe de définition, mais également par toutes les classes du même paquetage (**`package`**). En JAVA, un paquetage est une collection de classes placée dans des fichiers regroupés dans un même répertoire ou dossier, selon la terminologie du système d'exploitation utilisé. Un paquetage regroupe des classes qui possèdent des caractéristiques communes, comme par exemple le

6. Applications Programming Interface.

paquetage `java.io` pour toutes les entrées/sorties. La directive **`import`** suivie du nom d'un paquetage permet de dénoter les noms que définit ce dernier, sans les préfixer par le nom du paquetage. Par exemple, les deux déclarations de variables suivantes sont équivalentes :

```
import java.util;      ou alors      java.util.Random x;
Random x;
```

Des règles de visibilité sont également définies pour les classes. Une déclaration d'une classe préfixée par le mot-clé **`public`** rendra la classe visible par n'importe quelle classe depuis n'importe quel paquetage. Si ce mot-clé n'apparaît pas, la visibilité de la classe est alors limitée au paquetage. Les classes dont on veut limiter la visibilité au paquetage sont, en général, des classes auxiliaires nécessaires au bon fonctionnement des classes publiques du paquetage, mais inutiles en dehors.

Chapitre 13

Les exceptions

L'exécution anormale d'une action peut provoquer une erreur de fonctionnement d'un programme. Jusqu'à présent, lorsqu'une situation anormale était détectée, les programmes que nous avons écrits signalaient le problème par un message d'erreur et s'arrêtaient. Cette façon d'agir est pour le moins brutale, et expéditive. Dans certains cas, il serait souhaitable qu'ils reprennent le contrôle afin de poursuivre leur exécution. Les *exceptions* offrent une solution à ce problème.

13.1 ÉMISSION D'UNE EXCEPTION

Une exception est un événement qui indique une situation anormale, pouvant provoquer un dysfonctionnement du programme. Son origine est très diverse. Il peut s'agir d'exceptions matérielles, par exemple lors d'une lecture ou d'une écriture sur un équipement externe défectueux, ou encore d'une allocation mémoire impossible car l'espace mémoire est insuffisant. Ce type d'exception n'est pas directement de la responsabilité du programme (ou du programmeur). En revanche, les exceptions logicielles le sont, comme par exemple, une division par zéro ou l'indexation d'un composant de tableau en dehors du domaine de définition du type des indices. Plus généralement, le non respect des spécifications d'un programme (antécédents, conséquents, invariants de boucle ou de classe) provoque des exceptions logicielles.

Lorsqu'une exception signale le mauvais déroulement d'une action, cette dernière arrête le cours normal de son exécution. L'exception est alors prise en compte ou non. Si elle ne l'est pas le programme s'arrête définitivement. Bien souvent, ceci n'est pas acceptable, et en principe, le comportement habituel est, si possible, de traiter l'exception afin de poursuivre un déroulement normal du programme, c'est-à-dire en respectant ses spécifications.

13.2 TRAITEMENT D'UNE EXCEPTION

On distingue couramment deux façons de traiter une exception qui se produit au cours de l'exécution d'une action.

- La première consiste à exécuter à nouveau l'action en changeant les conditions initiales de son exécution. Il s'agit donc de changer les données de l'action, ou même de changer son algorithme. L'action peut être réexécutée une ou plusieurs fois jusqu'à ce que son conséquent final soit atteint.
- La seconde méthode consiste à transmettre l'exception à l'environnement d'exécution de l'action. S'il le peut, ce dernier traitera l'exception, ou alors la transmettra à son propre environnement. Les environnements d'exécution sont en général des contextes d'appel de sous-programmes. La figure 13.1 montre une chaîne d'appels de fonctions ou de procédures, symbolisée par les flèches pleines, depuis l'environnement initial E_1, jusqu'à un environnement E_4 dans lequel se produit une exception.

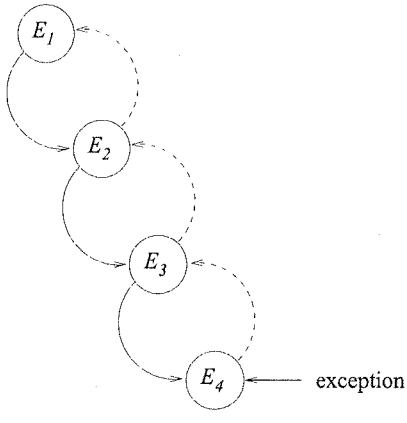


Figure 13.1 – *Une chaîne d'appels.*

Les flèches en pointillé indiquent les transmissions possibles de l'exception aux environnements d'appel. Notez que la chaîne des appels est parcourue en sens inverse. Chaque environnement peut traiter ou transmettre à l'environnement précédent l'exception. Si, en dernier ressort, l'exception n'est pas traitée par l'environnement initial E_1, le support d'exécution se charge d'arrêter le programme après avoir assuré diverses tâches de terminaison (fermetures de fichiers, par exemple).

La plupart des langages de programmation qui possèdent la notion d'exception proposent des mécanismes qui permettent de mettre en œuvre ces deux méthodes de gestion d'une exception. En revanche, à notre connaissance, seul EIFFEL inclut le concept de réexécution (instruction `retry`). D'autre part, son modèle d'exception est étroitement lié avec celui de la programmation contractuelle du langage [Mey97].

De façon plus formelle, l'objectif du traitement d'une exception est de maintenir la cohérence du programme. Dans le cas d'une programmation par objets, le traitement de l'exception devra maintenir l'invariant de classe et le conséquent de la méthode dans laquelle s'est

produite l'exception. Si l'exception est transmise à l'environnement d'appel, seul le maintien de l'invariant de classe est nécessaire.

Nous allons présenter maintenant la façon dont les exceptions sont gérées dans le langage JAVA.

13.3 LE MÉCANISME D'EXCEPTION DE JAVA

Une exception est décrite par un objet, occurrence d'une sous-classe de la classe `Throwable`. Cette classe possède deux sous-classes directes. La première, la classe `Error`, décrit des *erreurs systèmes* comme par exemple l'absence de mémoire. Ces exceptions ne sont en général pas traitées par les programmes. La seconde, la classe `Exception`, décrit des exceptions logicielles à traiter lorsqu'elles surviennent. Issues de ces deux classes, de nombreuses exceptions sont prédéfinies par l'API. Un programmeur peut également définir ses propres exceptions par simple héritage. La classe `Throwable` possède deux constructeurs que la nouvelle classe peut redéfinir.

```
class MonException extends Exception {
   public MonException () {
        ...
   }
   public MonException (String s) {
        ...
   }
}
```

13.3.1 Traitement d'une exception

Pour traiter une exception produite par l'exécution d'une action A, il faut placer la méthode dans une clause **`try`**, suivie obligatoirement d'une clause **`catch`** qui contient le traitement de l'exception.

```
try {
   A
}
catch (UneException e) {
   B
}
```

Si l'action A contenue dans la clause **`try`** du fragment de code précédent détecte une anomalie qui émet une exception de type `UneException`, son exécution est interrompue. Le programme se poursuit par l'exécution de l'action B placée dans la clause **`catch`** associée à l'exception `UneException`. Si aucune situation anormale n'a été détectée, l'exception `NomException` n'a donc pas été émise, l'action A est exécutée intégralement et l'action B ne l'est pas du tout. Dans la clause **`catch`**, `e` désigne l'exception qui a été récupérée et qui peut être manipulée par B. La méthode suivante contrôle la validité d'une valeur entière lue sur l'entrée standard. Si la lecture produit l'exception `IOException` (*e.g.* la valeur lue n'est

pas un entier), la clause **catch** capture l'exception et propose une nouvelle lecture. Notez que le nombre de lectures possibles n'est pas borné.

```
public int lireEntier() {
   try {
      return StdInput.readInt();
   }
   catch (IOException e) {
        System.out.println("réessayez : ");
        return lireEntier();
   }
}
```

Plusieurs clauses **catch**, associés à des exceptions différentes, peuvent suivre une clause **try**. Chacune des clauses **catch** correspond à la capture d'une exception susceptible d'être émise par les énoncés de la clause **try**. Si nécessaire, l'ordre des clauses **catch** doit respecter la hiérarchie d'héritage des exceptions.

Une clause **finally** peut également être placée après les clauses **catch**. Les énoncés qu'elle contient seront toujours exécutés qu'une exception ait été émise ou non, capturée ou non. Cette clause possède la forme suivante :

```
finally {
   énoncés
}
```

Une méthode qui contient une action susceptible de produire des exceptions n'est pas tenue de la placer dans une clause **try** suivie d'une clause **catch**. Dans ce cas, elle doit indiquer dans son en-tête, précédée du mot-clé **throws**, les exceptions qu'elle ne désire pas capturer. Si une exception apparaît, alors l'exécution se poursuit dans l'environnement d'appel de la méthode. Chaque méthode d'une chaîne d'appel peut traiter l'exception ou la transmettre à son environnement d'appel.

13.3.2 Émission d'une exception

L'émission explicite d'une exception est produite grâce à l'instruction **throw**. Le type de l'exception peut être prédéfini dans l'API, ou défini par le programmeur.

```
if (une situation anormale)
    throw new ArithmeticException();
if (une situation anormale)
    throw new MonException("un message");
```

Notez qu'une méthode (ou un constructeur) qui émet explicitement une exception doit également le signaler dans son en-tête avec le mot-clé **throws**, sauf si l'exception dérive de la classe `RuntimeException`.

13.4 EXERCICES

Exercice 13.1. Après chaque échec de lecture, la méthode `lireEntier` de la page 138 essaie une nouvelle lecture sans limiter le nombre de tentatives. Ceci peut être une source majeure de problèmes, si, par exemple, la saisie du nombre à lire est faite automatiquement par un programme qui produit systématiquement une valeur erronée. Modifiez la méthode `lireEntier` afin de limiter le nombre de tentatives, et de transmettre l'exception à l'environnement d'appel si aucune des tentatives n'a réussi.

Exercice 13.2. Écrivez une méthode qui calcule l'inverse d'un nombre réel x quelconque. Lorsque x est trop petit, l'opération produit une division par zéro, mais la méthode devra retourner dans ce cas la valeur zéro.

Exercice 13.3. Modifiez le constructeur de la classe `Date` donnée page 73 afin qu'il renvoie l'exception `DateException` si le jour, le mois et l'année ne correspondent pas à une date valide. Vous définirez une classe `DateException` pour représenter cette exception.

Chapitre **14**

Les fichiers séquentiels

Jusqu'à présent, les objets que nous avons utilisés dans nos programmes, étaient placés en mémoire centrale. À l'issue de l'exécution du programme ces objets disparaissaient. Cette réflexion appelle deux commentaires: 1) que faire si on désire manipuler des objets d'une taille supérieure à la mémoire centrale? 2) que faire si on veut conserver ces données après la fin de l'exécution du programme?

Les fichiers sont une réponse à ces deux questions. Il est très important de comprendre que ce concept de fichier, propre à un langage de programmation donné, trouve sa réalisation effective dans les mécanismes d'entrées-sorties avec le monde extérieur grâce aux dispositifs périphériques de l'ordinateur.

Il existe plusieurs modèles de fichier. Celui que nous étudierons, et que tous les langages de programmation mettent en œuvre, est le modèle *séquentiel*. Les fichiers séquentiels jouent un rôle essentiel dans tout système d'exploitation d'ordinateur. Ils constituent la structure appropriée pour conserver des données sur des dispositifs périphériques, pour lesquels les méthodes de lecture ou d'écriture des données passent dans un ordre strictement séquentiel (imprimantes, lecteurs de bandes, mais aussi, si on le désire, disques et disquettes).

Les fichiers séquentiels modélisent la notion de *suite* d'éléments. Quelle est la signification du qualificatif séquentiel? Il signifie que l'on ne peut accéder à un composant qu'*après* avoir accédé à *tous* ceux qui le précédent. Lors du traitement d'un fichier séquentiel, à un moment donné, *un seul* composant du fichier est accessible, celui qui correspond à la *position courante* du fichier. Les opérations définies sur les fichiers séquentiels permettent de modifier cette position courante pour accéder au composant suivant.

14.1 DÉCLARATION DE TYPE

Un objet de type *fichier séquentiel* forme une *suite* de composants tous de même type `T`, élémentaire ou structuré. La déclaration est notée :

```
fichier de T
```

Le type `T` des éléments peut être n'importe quel type à l'exception du type fichier ou de tout type structuré dont un composant serait de type fichier. En d'autres termes, les fichiers de fichiers ne sont pas autorisés.

Le nombre de composants n'est pas fixé par cette déclaration, c'est-à-dire que le domaine des valeurs d'un objet de type fichier est (théoriquement) infini.

Par exemple, les déclarations suivantes définissent deux variables `f1` et `f2`, respectivement, de type fichier d'entiers et de rectangles :

```
variables
    f1 type fichier de entier
    f2 type fichier de Rectangle
```

Les fichiers séquentiels se prêtent à deux types de manipulation : la lecture et l'écriture. Nous considérerons par la suite que ces deux manipulations ne peuvent avoir lieu en même temps ; un fichier est soit en lecture, soit en écriture, mais pas les deux à la fois.

14.2 NOTATION

Par la suite, nous utiliserons les notations suivantes qui nous permettront d'exprimer l'antécédent et le conséquent des opérations de base sur les fichiers.

Une variable `f` de type **`fichier de`** `T` est la concaténation de tous les éléments qui précèdent la position courante, désignés par $\overleftarrow{f}$, et de tous les éléments qui suivent la position courante, désignés par $\overrightarrow{f}$. L'élément courant situé à la position courante est noté $f\uparrow$

$$f = \overleftarrow{f} \,\&\, \overrightarrow{f}$$

$$f\uparrow = \text{premier}(\overrightarrow{f})$$

Une suite de composants de fichier est notée entre les deux symboles < et >. Par exemple, `<5 -3 10>` définit une suite de 3 entiers, et <> la suite vide.

La déclaration de type fichier n'indique pas le nombre de composants. Ce nombre est quelconque. Nous verrons plus loin qu'il nous sera nécessaire, lors de la manipulation des fichiers, de savoir si nous avons atteint la fin du fichier ou pas. Pour cela, nous définissons la fonction `fdf`, pour *fin de fichier*, qui renvoie vrai si la fin de fichier est atteinte et faux sinon. La signature de cette fonction est :

`fdf` : Fichier → booléen

Notez que :

$$\text{fdf}(f) \Rightarrow \overrightarrow{f} = <>$$

14.3 MANIPULATION DES FICHIERS

Les deux grandes formes de manipulation des fichiers sont l'écriture et la lecture. La première est utilisée pour la création des fichiers, la seconde pour leur consultation.

14.3.1 Écriture

Nous nous servirons des opérations d'écriture chaque fois que nous aurons à créer des fichiers. Pour créer un fichier, il est nécessaire d'effectuer au préalable une initialisation grâce à la procédure `InitÉcriture`. Son effet sur un fichier `f` est donné par :

```
{} InitÉcriture(f) {f = <> et fdf(f)}
```

L'initialisation en écriture d'un fichier a donc pour effet de lui affecter une suite de composants vide et peu importe si le fichier contenait des éléments au préalable.

La procédure `écrire` ajoute un composant à la fin du fichier. Remarquez que le prédicat `fdf(f)` est toujours vrai.

```
{f = x, e = t∈T et fdf(f)}}
écrire(f,e)
{f = x & <t> et  fdf(f)}
```

À partir de ces deux opérations, nous pouvons donner le schéma de la création d'un fichier :

```
Algorithme création d'un fichier
    {initialisation}
    InitÉcriture(f)
    tantque B faire
        {calculer un nouveau composant}
        calculer(e)
        {l'écrire à la fin du fichier}
        écrire(f,e)
    fintantque
```

L'expression booléenne `B` est un prédicat qui contrôle la fin de création du fichier.

Donnons, par exemple, l'algorithme de création un fichier de `n` réels tirés au hasard avec la fonction `random`. À l'aide du modèle précédent, nous écrirons :

```
Algorithme création d'un fichier d'entiers
    donnée n type naturel
    résultat f type fichier de réel
    {Antécédent : n ⩾ 0}
    {Conséquent : i=n et f contient n réels tirés au hasard}
    InitÉcriture(f)
    i ← 0
    tantque i ≠ n faire
        i ← i + 1
        écrire(f,random())
    fintantque
```

14.3.2 Lecture

Une fois le fichier créé, il peut être utile de consulter ses composants. La consultation d'un fichier débute par une initialisation en lecture grâce à la procédure `InitLecture`. Pour décrire son fonctionnement, nous distinguerons le cas où le fichier est initialement vide et le cas où il ne l'est pas.

```
{f = <>}
InitLecture(f)
{fdf(f) et ←f = →f = <>}

{f = x}
InitLecture(f)
{f = →f, ←f = <> et non fdf(f) et f↑=premier(x)}
```

Notez que l'initialisation en lecture d'un fichier non vide a pour effet d'affecter à la variable tampon la première valeur du fichier.

L'opération de lecture, `lire`, renvoie la valeur de l'élément courant et change la position courante (*i.e.* passe à la suivante). Nous allons distinguer le cas où l'élément à lire est le dernier du fichier et le cas où il ne l'est pas.

```
{←f = x et →f = <t> et non fdf(f) et f↑ = t}
e ← lire(f)
{←f = x & <t> et →f = <> et fdf(f) et e = t}

{f = x et →f = <t> & y et non fdf(f) et f↑ = t}
e ← lire(f)
{←f = x & <t> et →f = y et f↑ = premier(y) et non fdf(f) et e=t}
```

Notez que toute tentative de lecture *après* la fin de fichier est bien souvent considérée par les langages de programmation comme une erreur.

Le schéma général de consultation d'un fichier est donné par l'algorithme suivant :

```
Algorithme consultation d'un fichier
    {initialisation}
    InitLecture(f)
    tantque non fdf(f) faire
        {f↑ est l'élément courant du fichier lu}
        e ← lire(f)
        traiter(e)
    fintantque
```

Nous désirons écrire un algorithme qui calcule la moyenne des éléments du fichier de réels que nous avons créé plus haut.

```
Algorithme moyenne
    {Antécédent : f fichier de réels}
    {Conséquent : moyenne = moyenne des réels contenus dans
                  le fichier f ; si f est vide ⇒ moy = 0}
```

```
variables
     nbélt type naturel
     moyenne type réel

InitLecture(f)
moyenne ← 0
nbélt ← 0
tantque non fdf(f) faire
     {moyenne = ∑_{i=1}^{longueur(←f)} f_i et non fdf(f)}
     moyenne ← moyenne + lire(f)
     nbélt ← nbélt + 1
fintantque
si nbélt = 0 alors moyenne ← 0
     sinon moyenne ← moyenne/nbélt
finsi
rendre moyenne
```

14.4 LES FICHIERS DE JAVA

En JAVA, un *flot* (en anglais *stream*) est un support de communication de données entre une source émettrice et une destination réceptrice. Ces deux extrémités sont de toute nature ; ce sont par exemple des fichiers, la mémoire centrale, ou encore un programme local ou distant.

Les flots sont des objets définis par deux familles de classes. La première, représentée par les classes `InputStream` et `OuputStream`, sont des flots d'octets (8 bits) utilisés pour l'échange de données de forme quelconque. La seconde, représentée par les classes `Reader` et `Writer`, définit des flots de caractères Unicode (codés sur 16 bits) qui servent en particulier à la manipulation de texte.

Le paquetage `java.io` contient toute la hiérarchie des classes de flots qui assurent toutes sortes d'entrée-sortie. Leur description complète n'est pas du ressort de cet ouvrage. Nous ne présenterons dans cette section que les classes qui permettent la manipulation séquentielle de fichiers de données, élémentaires et structurées. Nous traiterons le cas particulier des fichiers de texte à la fin du chapitre.

14.4.1 Fichiers d'octets

On utilise les fichiers d'octets pour manipuler de l'information non structurée, ou du moins dont la structure est sans importance pour le traitement à effectuer. La déclaration et l'ouverture d'un fichier en lecture, respectivement en écriture, est faite avec la création d'un objet de type `FileInputStream`, respectivement `FileOutputStream` :

```
FileInputStream is = new FileInputStream("entrée");
FileOutputStream os = new FileOutputStream("sortie");
```

Ces classes offrent plusieurs constructeurs. Celui de l'exemple précédent admet comme donnée une chaîne de caractères qui représente un nom de fichier. Il est également possible

de lui fournir un descripteur d'un fichier déjà ouvert, ou un objet de type `FILE` (une représentation des noms des fichiers indépendante du système d'exploitation).

Parmi les méthodes proposées par les classes `FileInputStream` et `FileOutputStream`, il faut retenir plus particulièrement les méthodes :

- `read` de `FileInputStream` qui lit le prochain octet du fichier et le renvoie sous forme d'un entier ; cette fonction renvoie l'entier `-1` lorsque la fin du fichier est atteinte ;
- `write` de `FileOutputStream` qui écrit son paramètre (de type un **`byte`**) sur le fichier ;
- `close` qui ferme le fichier.

La méthode suivante assure une copie de fichier sans se préoccuper de son contenu.

```
// copie du fichier source dans le fichier destination
public void copie (String source, String destination)
throws IOException
{
     FileInputStream is = new FileInputStream(source);
     FileOutputStream os = new FileOutputStream(destination);
     int c;
     while ((c = is.read()) != -1)
          // os = is
          os.write((byte) c);
     // fin de fichier de is
     // fermer les fichiers is et os
     is.close();
     os.close();
}
```

Notez que cette méthode signale la transmission possible d'une exception `IOException` qui peut être déclenchée par les constructeurs en cas d'erreur d'ouverture des fichiers.

14.4.2 Fichiers d'objets élémentaires

Les classes `DataInputStream` ou `DataOutputStream` permettent de structurer, en lecture ou en écriture, des fichiers d'octets en fichiers de type élémentaire. Les constructeurs de ces deux classes attendent donc des objets de type `FileInputStream` ou `FileOutputStream`.

```
DataInputStream is =
          new DataInputStream(new FileInputStream("entrée"));
DataOutputStream os =
          new DataOutputStream(new FileOutputStream("sortie"));
```

Ces classes fournissent des méthodes spécifiques selon la nature des éléments à lire ou écrire (entiers, réels, caractères, *etc.*). Le nom des méthodes est composé de `read` ou `write` suffixé par le nom du type élémentaire (*e.g.* `readInt`, `writeInt`, `readFloat`, `writeFloat`, etc.).

Les opérations de lecture émettent l'exception `EOFException` lorsque la fin de fichier est atteinte. Le traitement de l'exception placée dans une clause **`catch`** consistera en général

à fermer le fichier à l'aide de la méthode `close`. On peut regretter l'utilisation par le langage du mécanisme d'exception pour traiter la fin de fichier. Est-ce vraiment une situation anormale que d'atteindre la fin d'un fichier lors de son parcours ?

Programmons les deux algorithmes de création d'un fichier d'entiers tirés au hasard, et du calcul de leur moyenne. Nous définirons une classe *Fichier* qui contiendra un constructeur qui fabrique le fichier d'entiers, et une méthode `moyenne` qui calcule la moyenne. Cette classe possède un attribut privé, `nomFich`, qui est le nom du fichier.

```
class Fichier {
  private String nomFich;
  Fichier(String nom, int n) throws IOException
  // Rôle : crée un fichier de n entiers tirés au hasard
  {
     nomFich = nom;
     // créer un générateur de nombres aléatoires
     Random rand = new Random();
     DataOutputStream f =
          new DataOutputStream (new FileOutputStream(nomFich));
     for (int i=0; i<n; i++)
          f.writeInt(rand.nextInt());
     // le fichier f contient n réels tirés au hasard
     f.close();
  }
  public double moyenne() throws IOException
  // Rôle : retourne la moyenne des valeurs du fichier courant
  {
     DataInputStream f =
          new DataInputStream(new FileInputStream(nomFich));
     int nbélt=0, moyenne=0;
     try {
         while (true) {
            // moyenne = \sum_{i=1}^{longueur(\overleftarrow{f})} f_i et non fdf(f)
            moyenne += f.readInt();
            nbélt++;
         }
     }
     catch (EOFException e) {
         // fin de fichier de f
         f.close();
     }
     return nbélt==0 ? 0 : (double) moyenne/nbélt;
  }
} // fin classe Fichier
```

On désire maintenant *fusionner* deux suites croissantes d'entiers. Ces deux suites sont contenues dans deux fichiers, `f` et `g`. Le résultat de la fusion est une troisième suite elle-même croissante placée dans le fichier `h`.

```
f = -10  -2  0  5  89  100
g = -50  0  1
h = fusionner(f,g) = -50  -10  -2  0  0  1  5  89  100
```

L'algorithme lit le premier élément de chacun des fichiers f et g. Il parcourt ensuite les deux fichiers *simultanément* jusqu'à atteindre la fin de fichier de l'un ou de l'autre. Les éléments courants des deux fichiers sont chaque fois comparés. Le plus petit est écrit sur le fichier *h*. Sur le fichier qui le contenait, on lit le prochain entier. Lorsque la fin de l'un des deux fichiers est atteinte (remarquez qu'il n'est pas possible d'atteindre la fin des deux fichiers simultanément), les entiers restants sont écrits sur h, puisque supérieurs à tout ceux qui précèdent. Cet algorithme s'exprime formellement :

```
Algorithme fusionner
   données f, g type fichiers de entier
   résultat h type fichier de entier
   {Antécédent : f et g deux fichiers qui contiennent des
                 suites croissantes d'entiers}
   {Conséquent : h = suite croissante d'entiers résultat
                  de la fusion des suites de f et g}
      InitLecture(f) InitLecture(g) InitÉcriture(h)
      si non fdf(f) et non fdf(g) alors
         {les deux fichiers ne sont pas vides}
         x ← lire(f)
         y ← lire(g)
      finsi
      tantque non fdf(f) et non fdf(g) faire
         {mettre dans h min(f,g) et passer au suivant}
         si x ⩽ y alors
              écrire(h,x)
              x ← lire(f)
          sinon {x>y}
            écrire(h,y)
            y ← lire(g)
         finsi
      fintantque
      {fdf(f) xou fdf(g)}
      si fdf(f) alors
         {recopier tous les éléments de g à la fin de h}
         écrire(h,y)
         recopier(g,h)
      sinon {recopier tous les éléments de f à la fin de h}
         écrire(h,x)
         recopier(f,h)
      finsi
```

L'algorithme de la procédure de recopie ne possède aucune difficulté d'écriture :

```
procédure recopier(donnée f : fichier de entier
                   résultat g : fichier de entier
{Antécédent : f non vide et ouvert en lecture
              g ouvert en écriture}
{Rôle : recopie à la fin de g les éléments de f}
variable x type entier
```

```
   répéter
      x ← lire(f)
      écrire(g,x)
   jusqu'à fdf(f)
finproc {recopier}
```

Programmons en JAVA cet algorithme avec la méthode `fusionner` qui complétera la classe `Fichier`. Cette méthode fusionne les deux fichiers passés en paramètre. L'objet courant contient le résultat de la fusion.

```
public void fusionner(String f1, String f2)
// Antécédent: f1 et f2 deux noms de fichiers qui contiennent des
// suites croissantes d'entiers
// Conséquent: le fichier courant nomFich est la suite croissante
// d'entiers résultat de la fusion des suites de f1 et f2
throws IOException, EOFException {
  DataInputStream f =
                  new DataInputStream(new FileInputStream(f1));
  DataInputStream g =
                  new DataInputStream(new FileInputStream(f2));
  DataOutputStream h =
           new DataOutputStream(new FileOutputStream(nomFich));
  int x, y;
  // lire le premier entier de chacun des fichiers
  try { x = f.readInt(); }
  catch (EOFException e) { // fdf(f) ⇒ recopier g sur h
         recopier(g,h);
         return;
  }
  try { y = g.readInt();  }
  catch (EOFException e) { // fdf(g) ⇒ recopier x et f sur h
         h.writeInt(x);
         recopier(f,h);
         return;
  }
  // les fichiers h et g contiennent tout deux au moins un entier
  while (true)
     // mettre dans h min(f,g) et passer au suivant
     if (x<=y) { // écrire x sur h
        h.writeInt(x);
        try { x = f.readInt(); }
        catch (EOFException e) {
               // fdf(f) ⇒ recopier y et g sur h
               h.writeInt(y);
               recopier(g,h);
               return;
        }
     }
     else { // x>y ⇒ écrire y sur h
            h.writeInt(y);
```

```
            try { y = g.readInt(); }
            catch (EOFException e) {
                    // fdf(g) ⇒ recopier x et f sur h
                    h.writeInt(x);
                    recopier(f,h);
                    return;
            }
      }
   }
} // fin fusionner
```

Vous remarquerez que la recopie des derniers entiers, lorsque la fin d'un des fichiers est détectée, est faite *dans* la boucle, et non pas après comme l'algorithme le suggère. Si une clause **catch** avait été placée après l'énoncé itératif, elle aurait attrapée l'exception `EOFException`, mais sans pouvoir en déterminer la provenance (*i.e.* fin de fichier de `f` ou de `g`).

La méthode `recopie` est déclarée privée, dans la mesure où elle n'est utilisée que dans la classe.

```
private void recopier(DataInputStream i, DataOutputStream o)
// Antécédent : fichier i non vide et ouvert en lecture
// fichier o ouvert en écriture
// Rôle : recopie à la fin de o les éléments de i
{
  try {
      while (true) o.writeInt(i.readInt());
   }
   catch (EOFException e) { // fdf(i)
            i.close(); o.close();
   }
}
```

14.4.3 Fichiers d'objets structurés

La lecture et l'écriture de données structurées sont faites en reliant des objets de type `FileInputStream` ou `FileOutputStream` avec des objets de type `ObjectInputStream` ou `ObjectOutputStream`. Ces classes fournissent respectivement les méthodes `readObject` et `writeObject` pour lire et écrire un objet. Le fragment de code suivant écrit dans le fichier `Frect` un objet de type `Rectangle` (défini au chapitre 7), puis le relit.

```
Rectangle r = new Rectangle(3,4);
ObjectOutputStream os =
                new ObjectOutputStream(new FileOutputStream("Frect"));
// écriture d'un objet de type Rectangle sur le fichier os
os.writeObject(r);
os.close();
ObjectInputStream is =
                new ObjectInputStream(new FileInputStream("Frect"));
```

```
// lecture d'un objet de type Rectangle sur le fichier is
r = (Rectangle) is.readObject();
is.close();
```

Notez que la méthode `readObject` retourne des objets de type `Object` qui doivent être explicitement convertis si la règle de compatibilité de type l'exige. Dans notre exemple, la conversion de l'objet lu en `Rectangle` est imposée par le type de la variable `r`[1]. Si le type de conversion n'est pas celui de l'objet lu, il se peut alors que l'erreur ne soit découverte qu'à l'exécution. Par exemple, si l'objet lu est de type `Rectangle`, le fragment de code suivant ne produit aucune erreur de compilation.

```
Integer z = (Integer) f.readObject();
System.out.println(z);
```

De plus, la méthode `readObject` émet l'exception `EOFException` si la fin de fichier est atteinte. Enfin, les classes qui définissent des objets qui peuvent être écrits et lus sur des fichiers doivent spécifier dans leur en-tête qu'elles implantent l'interface `Serializable`. L'en-tête de la classe `Rectangle` s'écrit alors :

```
public class Rectangle implements Serializable {
   ...
}
```

Si elles ne le font pas, les méthodes `writeObject` et `readObject` émettront, respectivement, les exceptions `NotSerializableException` et `ClassNotFoundException`.

14.5 LES FICHIERS DE TEXTE

Les fichiers de texte jouent un rôle fondamental dans la communication entre le programme et les utilisateurs humains. Ces fichiers ont des composants de type *caractère* et introduisent une structuration en ligne. Un fichier de texte est donc vu comme une suite de lignes, chaque ligne étant une suite de caractères quelconques terminée par un caractère de fin de ligne. Certains langages comme le langage PASCAL définissent même un type spécifique pour les représenter. Pour d'autres langages, ils sont simplement des fichiers de caractères.

Alors que les éléments de ces fichiers de texte sont des caractères, bien souvent les langages de programmation autorisent l'écriture et la lecture d'éléments de types différents, mais qui imposent une conversion *implicite*. Par exemple, il sera possible d'écrire ou de lire un entier sur ces fichiers. L'écriture de l'entier 125 provoque sa conversion implicite en la suite de trois caractères `'1'`, `'2'` et `'5'` successivement écrits dans le fichier de texte. Il est important de bien comprendre que les fichiers de texte ne contiennent que des caractères, et que la lecture ou l'écriture d'objet de type différent entraîne une conversion de type depuis ou vers le type caractère.

La plupart des langages de programmation définissent des fichiers de texte liés *implicitement* au clavier et à l'écran de l'ordinateur. Ces fichiers sont appelés fichier d'*entrée standard*

1. Le compilateur signale une erreur si la conversion n'est pas explicitement faite.

et fichier de *sortie standard*. Certains langages proposent un fichier de sortie d'erreur standard dont les programmes se servent pour écrire leurs messages d'erreurs. Le fichier d'entrée standard ne peut évidemment être utilisé qu'en lecture, alors que ceux de sortie standard et de sortie d'erreur standard ne peuvent l'être qu'en écriture. Ces fichiers sont toujours automatiquement ouverts au démarrage du programme.

14.6 LES FICHIERS DE TEXTE EN JAVA

Les flots de texte sont construits à partir des sous-classes issues des classes `Reader`, pour les flots d'entrée, et `Writer`, pour ceux de sortie. Le type des éléments est le type **`char`** (UNICODE). Les classes `FileReader` et `FileWriter` permettent de définir les fichiers de caractères, mais la structuration en ligne des fichiers de texte n'est pas explicitement définie. Toutefois, le caractère `'\n'` permet de repérer la fin d'une ligne.

Écrivons le programme qui compte le nombre de caractères, de mots et de lignes contenus dans un fichier de texte[2]. On considère les mots comme des suites de caractères séparés par des espaces, des tabulations ou des passages à la ligne.

La seule petite difficulté de ce problème vient de ce qu'il faut reconnaître les mots dans le texte. Il est résolu simplement à l'aide d'un petit automate à deux états, *dansMot* et *horsMot*. Si l'état courant est *dansMot* et le caractère courant est un séparateur, l'état devient *horsMot* et on incrémente le compteur de mots puisqu'on vient d'achever la reconnaissance d'un mot. Si l'état courant est *horsMot* et le caractère courant n'est pas un séparateur, l'état devient *dansMot*. Notez que l'incrémentation du compteur de mot aurait pu tout aussi bien se faire au moment de la reconnaissance du premier caractère d'un nouveau mot. Le tableau suivant résume les changements d'états et les actions à effectuer.

état \ c	séparateur	non séparateur
dansMot	état←*horsMot* nbmot←nbmot+1	
horsMot		état←*dansMot*

Algorithme wc

```
variables
  f type fichier de texte
  état type (dansMot, horsMot)
  c type caractère {le caractère courant}

  état ← horsMot
  tantque non fdf(f) faire
    {lire le prochain caractère de f}
```

2. À l'instar de la commande wc du système d'exploitation UNIX.

```
        c ← lire(f)
        {incrémenter le compteur de caractères}
         nbcar ← nbcar+1
         si c est un séparateur alors
             si état = dansMot alors
                   {fin d'un mot ⇒ incrémenter le compteur de mots}
                   nbmots ← nbmots+1
                   état ← horsMot
             finsi
             si c = fin de ligne alors
                 {fin d'une ligne ⇒ incrémenter le compteur de lignes}
                 nblignes ← nblignes+1
             finsi
         sinon
             {c est un caractère d'un mot}
             si état = horsMot alors état ← dansMot finsi
         finsi
      fintantque
      {fin de fichier ⇒ afficher les résultats}
      écrire(nbLignes, nbMots, nbCar)
```

La programmation de cet algorithme est donnée ci-dessous. Dans la mesure où JAVA ne définit pas de constructeur de type énuméré, nous représentons la variable d'état par une valeur booléenne. Notez que la fin d'un fichier de type `Filereader` est détectée lorsque la valeur renvoyée par la méthode `read` est égale à `-1`, et que ceci impose que le type de la variable `c` soit le type entier **`int`**. Pour la traiter comme un caractère, il faut alors la convertir explicitement dans le type **`char`**.

```
import java.io.*;
public class Wc {
  public static void main(String [] args) throws IOException {
    if (args.length != 1) {
        // Wc attend un seul nom de fichier
        System.err.println("Usage: java Wc fichier");
        System.exit(1);
    }
    FileReader is = null;
    try {
          is = new FileReader(args[0]);
    }
    catch (FileNotFoundException e) {
       System.err.println("fichier '" +args[0]+ "' non trouvé");
       System.exit(2);
    }
    boolean étatDansMot=false;
    int c, nbCar=0, nbMots=0, nbLignes=0;
    while ((c=is.read()) != -1) {
        // incrémenter le compteur de caractères
        nbCar++;
```

```
        if (Character.isWhitepace((int) c)) {
            // c est un séparateur de mot
            if (étatDansMot) {
                // fin d'un mot ⇒ incrémenter le compteur de mots
                nbMots++;
                étatDansMot=false;
            }
            if (c=='\n')
              // fin d'une ligne ⇒ incrémenter le compteur de lignes
              nbLignes++;
        }
        else // on est dans un mot
            if (!étatDansMot) étatDansMot=true;
    }
    // fin de fichier ⇒ afficher les résultats
    System.out.println(nbLignes + "␣" + nbMots + "␣" + nbCar);
  }
} // fin classe Wc
```

Vous avez remarqué l'utilisation, pour la première fois, du paramètre `args` de la méthode `main`. Ce tableau contient les paramètres de commande passés au moment de l'exécution du programme. `Wc` attend un seul paramètre, le nom du fichier sur lequel le comptage sera effectué, contenu dans la chaîne de caractères `args[0]`. L'ouverture en lecture du fichier est placée dans une clause **`try`** afin de vérifier qu'il est bien lisible.

Les classes `FileReader` et `FileWriter` ne permettent de manipuler que des caractères. L'API de JAVA propose la classe `PrintWriter`, à connecter avec `FileWriter`, pour écrire des objets de n'importe quel type, après une conversion implicite sous forme d'une chaîne de caractères. Pour les objets non élémentaires, la conversion est assurée par la méthode `toString`. Cette classe propose essentiellement deux méthodes d'écriture, `print` et `println`. Cette dernière écrit en plus le caractère de passage à la ligne.

Le fragment de code suivant recopie un fichier `source` dans un fichier `destination` en numérotant les lignes. La valeur entière du compteur de ligne est écrite en début de ligne grâce à la méthode `print`.

```
FileReader is=new FileReader(source);
PrintWriter os=new PrintWriter(new FileWriter(destination));
int c, nbligne=1;
os.print(1 + "␣␣");
while ((c=is.read()) != -1) {
     os.write(c);
     if (c=='\n')
          // début d'une nouvelle ligne
          os.print(++nbligne + "␣␣");
}
is.close();
os.close();
```

Il est étonnant que l'API ne propose pas de classe équivalente à `PrintWriter` pour lire des objets de type quelconque à partir de leur représentation sous forme de caractères. Aucune

classe standard ne définit, par exemple, une méthode `readInt` qui lit une suite de chiffres sur un fichier de type `FileReader` pour retourner la valeur entière qu'elle représente. Pourtant une telle classe est très utile, et bien souvent, les programmeurs sont amenés à construire leur propre classe.

Le langage JAVA propose trois fichiers standard : l'entrée standard `System.in`, la sortie standard `System.out` et la sortie d'erreur standard `System.err`. Les méthodes offertes par `System.in` sont celles de la classe `InputStream`, et ne permettent de lire que des caractères sur huit bits ; aucune n'offre la possibilité de lire des objets de type quelconque après conversion. Les fichiers `System.out` et `System.err` de type `PrintStream` utilisent également des flots d'octets, mais la méthode `print` (ou `println`) permet l'écriture de n'importe quel type d'objet.

Notez que la classe `StdInput` que nous avons utilisée jusqu'à présent permet de lire sur l'entrée standard des objets de type élémentaire, mais ce n'est pas une classe standard de l'API. Elle a été développée par l'auteur pour des besoins pédagogiques.

Pour conclure, signalons l'existence des classes `InputStreamReader` et `OutputStreamWriter` qui sont des passerelles entre les flots d'octets et les flots de caractères UNICODE. La plupart du temps, les systèmes d'exploitation codent les caractères des fichiers de texte sur huit bits, et ces classes permettront la conversion d'un octet en un caractère UNICODE codé sur seize bits, et réciproquement. Notez que les instructions suivantes :

```
new FileReader(f)
new FileWriter(f)
```

sont strictement équivalentes à :

```
new InputStreamReader(new FileInputStream(f))
new OutputStreamWriter(new FileOutputStream(f))
```

14.7 EXERCICES

Exercice 14.1. Reprogrammez l'algorithme d'ÉRATOSTHÈNE de la page 94 en choisissant un fichier séquentiel d'entiers pour représenter le crible. Notez que vous aurez besoin d'un second fichier auxiliaire.

Exercice 14.2. Écrivez un programme qui supprime tous les commentaires d'un fichier contenant des classes JAVA. On rappelle qu'il existe trois formes de commentaires.

Exercice 14.3. Le fragment de code (donné à la page 154) qui recopie le contenu d'un fichier en numérotant les lignes a un léger défaut. Il numérote systématiquement une dernière ligne inexistante. Modifiez le code afin de corriger cette petite erreur.

Exercice 14.4. Rédigez une classe `FichierTexte` qui gère la notion de ligne des fichiers de texte. Vous définirez toutes les méthodes du modèle algorithmique de fichier donné dans ce chapitre, complétées par les fonctions `fdln`, `lireln` et `écrireln`. La fonction `fdln` retourne un booléen qui indique si la fin d'une ligne est atteinte ou pas. La procédure `lireln`

fait passer à la ligne suivante, et la procédure `écrireln` écrit la marque de fin de ligne. Avec ces nouvelles fonctions, l'algorithme de consultation d'un fichier de texte a la forme suivante :

```
{initialisation}
InitLecture(f)
tantque non fdf(f) faire
    {on est en début de ligne, traiter la ligne courante}
    tantque non fdln(f) faire
       c ← lire(f)
       traiter(c) {traiter le caractère courant}
    fintantque
    {on est en fin de ligne, passer à la ligne suivante}
    lireln(f)
fintantque
```

Exercice 14.5. Complétez la classe `FichierTexte` afin de permettre des lectures et des lectures des objets de type quelconque. Pour les lectures, vous vous limiterez aux types élémentaires.

Chapitre **15**

Récursivité

Les fonctions récursives jouent un rôle très important en informatique. En 1936, avant même l'avènement des premiers ordinateurs électroniques, le mathématicien A. CHURCH[1] avait émis la thèse[2] que toute fonction calculable, c'est-à-dire qui peut être résolue selon un algorithme sur une machine, peut être décrite par une fonction récursive.

Dans la vie courante, il est possible de définir un oignon comme de la pelure d'oignon qui entoure un oignon. De même, une matriochka est une poupée russe dans laquelle est contenue une matriochka. Ces deux définitions sont dites *récursives*. On parle de définition récursive lorsqu'un terme est décrit à partir de lui-même. En mathématique, certaines fonctions, comme par exemple la fonction factorielle $n! = n \times (n-1)!$, sont également définies selon ce modèle. On parle alors de définition par récurrence.

Les précédentes définitions de l'oignon et de la matriochka sont *infinies*, en ce sens qu'on ne voit pas comment elles s'achèvent, un peu comme quand on se place entre deux miroirs qui se font face, et qu'on aperçoit son image reproduite à l'infini. Pour être calculables, les définitions récursives ont besoin d'une *condition d'arrêt*. Un oignon possède toujours un noyau qu'on ne peut pas peler, il existe toujours une toute petite *matriochka* qu'on ne peut ouvrir, et enfin $0! = 1$.

Dans ce chapitre, nous aborderons deux sortes de récursivité : la récursivité des *actions* et celle des *objets*.

1. Mathématicien américain (1903-1995).

2. Non contredite jusqu'à aujourd'hui.

15.1 RÉCURSIVITÉ DES ACTIONS

15.1.1 Définition

Dans les langages de programmation, la récursivité des actions se définit par des fonctions ou des procédures récursives. Un sous-programme récursif contiendra au moins un énoncé d'appel, direct ou non, à lui-même dans son corps.

Une procédure (ou une fonction) récursive P, définie selon le modèle ci-dessous, crée un nombre infini d'*incarnations* de la procédure au moyen d'un nombre fini d'énoncés.

$$P = C(E_i,P)$$

où C représente une composition d'énoncés E_i.

15.1.2 Finitude

La définition précédente ne limite pas le nombre d'appels récursifs, et un programme écrit selon ce modèle ne pourra jamais s'achever. Il est donc nécessaire de limiter le nombre des appels récursifs. L'appel récursif d'une fonction ou d'une procédure devra *toujours* apparaître dans un énoncé conditionnel et s'appliquer à un sous-ensemble du problème à résoudre.

$$P = \texttt{si } B \texttt{ alors } C(E_i,P) \texttt{ finsi}$$

ou bien

$$P = C(E_i, \texttt{si } B \texttt{ alors } P \texttt{ finsi})$$

Toutefois, il faut être sûr que l'exécution des instructions conduira tôt ou tard à la branche de l'énoncé conditionnel qui ne contient pas d'appel récursif. Comme pour les énoncés itératifs, il est essentiel de prouver la *finitude* du sous-programme P. Pour cela, on lui associe un ou plusieurs paramètres qui décrivent le domaine d'application du sous-programme. Les valeurs de ces paramètres doivent évoluer pour restreindre le domaine d'application et tendre vers une valeur particulière qui arrêtera les appels récursifs. Le modèle devient alors :

$$P_x = \texttt{si } B \texttt{ alors } C(E_i,P_{x'}) \texttt{ finsi}$$

ou bien

$$P_x = C(E_i, \texttt{si } B \texttt{ alors } P_{x'} \texttt{ finsi})$$

où x' est une partie de x. Bien sûr, x et x' ne peuvent être égaux, sinon, en dehors de tout effet de bord, les appels récursifs seraient tous identiques et le sous-programme ne pourrait s'achever.

15.1.3 Écriture récursive des sous-programmes

Pour beaucoup de néophytes, l'écriture récursive des sous-programmes apparaît être une réelle difficulté, et bien souvent ceux-ci envisagent *a priori* une solution non récursive, c'est-à-dire basée sur un algorithme itératif. Pourtant, certains pensent que l'itération est humaine

et la récursivité divine. Au delà de cet aphorisme, l'écriture récursive est immédiate pour des algorithmes *naturellement* récursifs, comme les définitions mathématiques par récurrence, ou pour ceux qui manipulent des objets récursifs (voir 15.2). Elle est aussi plus concise, et souvent plus claire que son équivalent itératif, même si sa compréhension nécessite une certaine habitude. De plus, l'analyse de la complexité des algorithmes récursifs est souvent plus simple à mettre en œuvre.

Les définitions par récurrence des fonctions mathématiques ont une transposition algorithmique évidente. Par exemple, les fonctions *factorielle* et *fibonacci*[3] s'expriment par récurrence comme suit :

$$\begin{aligned} \text{fac}(0) &= 1 \\ \text{fac}(n) &= n \times \text{fac}(n-1), \forall n > 0 \end{aligned}$$

$$\begin{aligned} \text{fib}(1) &= 1 \\ \text{fib}(2) &= 1 \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2), \forall n > 2 \end{aligned}$$

L'écriture des deux algorithmes est immédiate dans la mesure où il suffit de respecter la définition mathématique de la fonction :

```
fonction factorielle(donnée n : naturel) : nature
{Antécédent : n≥0}
{Rôle : calcule n! = n × n-1!, avec 0! = 1}
     si n=0 alors rendre 1
        sinon rendre n × factorielle(n-1)
     finsi
finfonc {factorielle}

fonction fibonacci(donnée n : naturel) : naturel
{Antécédent : n>0}
{Rôle : calcule fib(n)= fib(n-1) + fib(n-2), pour n > 2
        et avec fib(1) = 1 et fib(2) = 1}
   si n≤2 alors rendre 1
     sinon rendre fibonacci(n-1) + fibonacci(n-2)
   finsi
finfonc {fibonacci}
```

Vous remarquerez que la finitude de ces deux fonctions est garantie par leur paramètre qui décroît à chaque appel récursif, pour tendre vers un et zéro, valeurs pour lesquelles la récursivité s'arrête.

Le problème des tours de Hanoï, voir la figure 15.1, consiste à déplacer n disques concentriques empilés sur un premier axe A vers un deuxième axe B en se servant d'un troisième axe intermédiaire C. La règle exige qu'un seul disque peut être déplacé à la fois, et qu'un disque ne peut être posé que sur un axe vide ou sur un autre disque de diamètre supérieur. La

3. Cette fonction fut proposée en 1202 par le mathématicien italien LEONARDO PISANO (1175–1250), appelé FIBONACCI (une contraction de *filius Bonacci*), pour calculer le nombre annuel de couples de lapins que peut produire un couple initial, en supposant que tous les mois chaque nouveau couple produit un nouveau couple de lapins, qui deviendra à son tour productif deux mois plus tard.

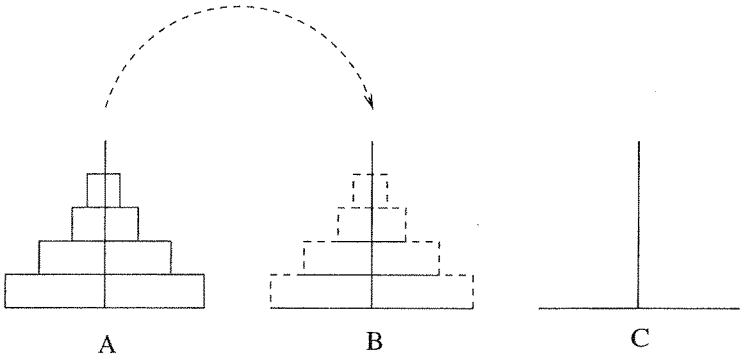


Figure 15.1 – *Les tours de Hanoï.*

légende indique que pour une pile de 64 disques et à raison d'un déplacement de disque par seconde, la fin du monde aura lieu lorsque la pile sera entièrement reconstituée ! Le nombre total de déplacements est exponentiel. Il est égal à $2^n - 1$.

L'écriture récursive de l'algorithme est très élégante et très concise. Pour placer, à sa position finale, le plus grand disque il faut que l'axe B soit vide, et qu'une tour, formée des $n-1$ restants, soit reconstituée sur l'axe C. Le déplacement de ces $n-1$ disques se fait bien évidemment par l'intermédiaire de l'axe B selon les règles des tours de Hanoï, c'est-à-dire de façon récursive. Il suffit ensuite de déplacer les $n-1$ disques de l'axe C vers l'axe B, toujours de façon récursive, en se servant de l'axe A comme axe intermédiaire.

```
procédure TroisdeHanoï(données n : nbdisques
                                a, b, c : axes)
{Rôle : déplacer n disques concentriques de l'axe a
       vers l'axe b, en utilisant c comme axe intermédiaire}
   si n>0 alors
       {déplacer n-1 disques de a vers c, intermédiaire b}
       TroisdeHanoï(n-1,a,c,b)
       {déplacer le disque n de l'axe a vers b}
       déplacer(n,a,b)
       {déplacer n-1 disques de c vers b, intermédiaire a}
       TroisdeHanoï(n-1,c,b,a)
   finsi
finproc {TroisdeHanoï}
```

Le nombre de disques déplacés récursivement diminue de un à chaque appel récursif et tend vers zéro. Pour cette valeur particulière la récursivité s'arrête, ce qui garantit la finitude de l'algorithme.

Prenons un dernier exemple. Nous désirons écrire une procédure qui écrit sur la sortie standard la suite de chiffres que forme un entier naturel. Imaginons que le langage ne mette à notre disposition qu'une procédure d'écriture d'un caractère et une fonction de conversion d'un chiffre en caractère. L'algorithme décompose le nombre par divisions entières successives et procède à l'écriture des chiffres produits.

```
procédure écrireChiffres(donnée n : naturel)
{Antécédent : n⩾0}
{Rôle : écrit sur la sortie standard la suite de
        chiffres qui forme l'entier n}
   si n⩾10 alors
        écrireChiffres(n/10)
   finsi
   écrire(convertirEnCaractère(n mod 10))
finproc {écrireChiffres}
```

La finitude de cet algorithme est assurée pour tout entier positif. Les appels récursifs s'appliquent à une suite d'entiers naturels qui tend vers un nombre inférieur à dix. Pour cette valeur, la récursivité s'arrête. Sa programmation en JAVA ne pose pas de difficulté particulière :

```
public static void écrireChiffres(int n)
// Antécédent : n⩾0
// Rôle : écrit sur la sortie standard la suite de
// chiffres qui forme l'entier n
{
    if (n>=10) écrireChiffres(n/10);
    // écrire la conversion du chiffre en caractère
    System.out.print((char) (n%10 + '0'));
}
```

15.1.4 La pile d'évalution

L'écriture itérative de la procédure `écrireChiffres` nécessite de mémoriser les chiffres (par exemple dans un tableau) parce que la décomposition par divisions successives produit les chiffres dans l'ordre inverse de celui souhaité. Le calcul du résultat est obtenu ensuite après un parcours à l'envers de la séquence de chiffres mémorisée.

```
public static void écrireChiffres(int n) {
   final int maxchiffres=10; // 32 bits / 3
   int[] chiffres=new int[maxchiffres];
   int i=0;
   // décomposer le nombre par divisions successives
   // mémoriser les chiffres dans le tableau
   do
        chiffres[i++]=n%10;
   while ((n/=10) != 0);
   // parcourir le tableau des chiffres en sens inverse
   // et écrire la conversion de chaque chiffre en caractère
   while (i>0)
      System.out.print((char) (chiffres[--i] + (int) '0'));
}
```

Pourquoi la version récursive se passe-t-elle du tableau ? En fait, la séquence d'appels récursifs mémorise les chiffres du nombre dans une pile « cachée », la pile d'évaluation du programme dans laquelle s'empilent les zones locales des procédures et des fonctions. Le

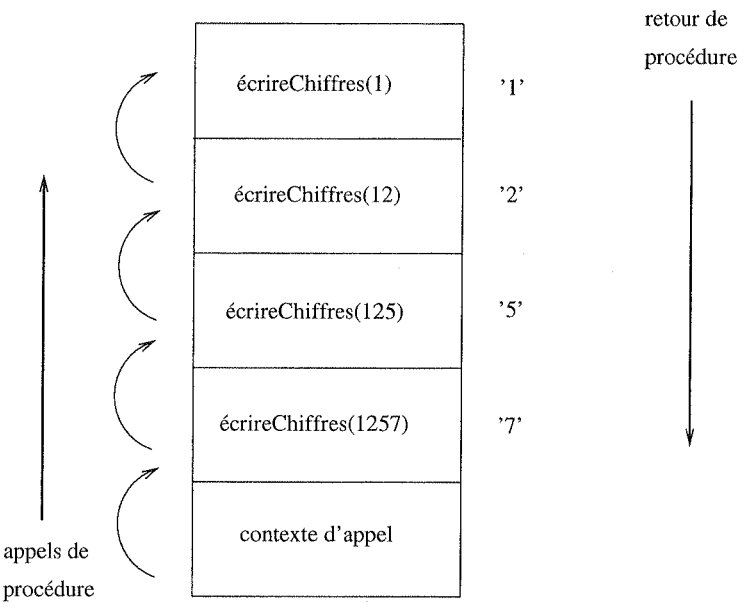


Figure 15.2 – *Exécution de écrireChiffres(1257).*

parcours à l'envers de la séquence de chiffres produite est obtenu *automatiquement* lorsqu'on dépile la pile d'évaluation, en fin d'exécution de chaque appel récursif. La figure 15.2 montre la pile d'évaluation pour l'appel de la fonction `écrireChiffres(1257)`. À gauche de la pile, les flèches indiquent l'empilement des zones locales produites par les appels récursifs et à sa droite est la valeur écrite par la procédure une fois l'exécution de chaque appel achevée.

De nombreux algorithmes récursifs se servent de la pile d'évaluation pour mémoriser des données, en particulier les paramètres transmis par valeur du sous-programme, et pour récupérer leur valeur lors du retour de l'appel récursif. La fonction `fibonacci` et la procédure `ToursDeHanoï` procèdent de la sorte. En exercice, vous pouvez essayer de *dérouler* à la main la suite des appels récursifs de ces fonctions, mais d'une façon générale, la compréhension d'un algorithme récursif doit se faire de façon synthétique à partir du cas général.

15.1.5 Quand ne pas utiliser la récursivité ?

Une première réponse à cette question est quand l'écriture itérative est *évidente*. Typiquement, c'est le cas pour les fonctions `factorielle` et `fibonacci` et on leur préférera les versions itératives suivantes :

```
fonction factorielle(donnée n : naturel) : naturel
{Rôle : calcule n! = n × n-1!, avec 0! = 1}
variables i, fact de type naturel
    i ← 0
    fact ← 1
```

```
    tantque i<n faire {fact = i!}
       i ← i+1
       fact ← fact × i
    fintantque
    {i=n et fact=n!}
    rendre fact
finfonc {factorielle}

fonction fibonacci(donnée n : naturel) : naturel
{Rôle : calcule fib(n)= fib(n-1) + fib(n-2), pour n>2
        et avec fib(1) = 1 et fib(2) = 1}
variables i, pred, succ de type naturel
   i ← 1
   pred ← 1
   succ ← 1
   tantque i<n faire
      {pred=fib(i) et succ=fib(i+1)}
      i ← i+1
      succ ← succ+pred
      pred ← succ-pred
   fintantque
  rendre pred
finfonc {fibonacci}
```

Deuxièmement, lorsque la récursivité est *terminale*, c'est-à-dire lorsque la dernière instruction du sous-programme est l'appel récursif. Le processus récursif est équivalent à un processus itératif à mettre en œuvre avec un énoncé **tantque**. Des sous-programmes de la forme :

$$P = \textbf{si } B \textbf{ alors } E\ P \textbf{ finsi}$$

$$P = E;\ \textbf{si } B \textbf{ alors } P \textbf{ finsi}$$

sont équivalents à :

$$P = \text{initialisation } \textbf{tantque } B \textbf{ faire } E \textbf{ fintantque}$$

L'écriture itérative de la fonction `factorielle` est une application directe de cette règle de transformation.

Troisièmement, quand l'efficacité en temps d'exécution des programmes est en jeu. La récursivité a un coût, celui des appels récursifs des procédures ou des fonctions. Pour s'en convaincre, comparez les temps d'exécution des versions itérative et récursive de `fibonacci` écrites en JAVA avec $n = 30$ et $n = 40$. Sur un Pentium II 266, pour la première valeur, la version itérative est 40 fois plus rapide que la version récursive, et 4000 fois plus rapide pour la seconde valeur. Le calcul récursif de `fibonacci` est particulièrement lourd : `fibonacci(5)` nécessite pas moins de 9 appels récursifs. Le nombre théorique d'appels est égal à $(2/\sqrt{5})1{,}618^n - 1$. La complexité de l'algorithmique récursif est exponentielle. On lui préférera donc celle linéaire de la version itérative.

D'une façon générale, et même si sur les ordinateurs actuels les appels des sous-programmes sont efficaces, lorsque le nombre d'appels récursifs devient supérieur à $n \log_2 n$, où n est le paramètre qui contrôle les appels récursifs, il est raisonnable de rechercher une solution itérative. De même, lorsque la profondeur de récursivité est supérieure à n, on recherchera une solution itérative.

Toutefois, il existe des situations où il est difficile de se prononcer parce que la solution itérative est loin d'être évidente. Regardons par exemple la fonction Ackermann[4] donnée par la relation de récurrence suivante :

$$\begin{aligned} \text{Ackermann}(0,n) &= n+1 \\ \text{Ackermann}(m,0) &= \text{Ackermann}(m-1,1) \\ \text{Ackermann}(m,n) &= \text{Ackermann}(m-1, \text{Ackermann}(m,n-1)) \end{aligned}$$

et dont la programmation en JAVA est donnée par :

```
public static long Ackermann(long m, long n) {
   if (m==0) return n+1;
   else
      if (n==0) return Ackermann(m-1,1);
      else
          return Ackermann(m-1,Ackermann(m,n-1));
}
```

Il est clair que cette écriture est très coûteuse en terme de nombre d'appels récursifs. La fonction `Ackermann` croît de façon exponentielle, mais plus rapidement encore que `fibonacci`. Toutefois, l'écriture récursive est immédiate, contrairement à la version itérative. En exercice, je vous laisse chercher la solution itérative. Sachez que pour toute fonction récursive, il existe toujours une solution itérative, ne serait-ce qu'en simulant la pile des appels récursifs. Notez bien que cette fonction ne contient que deux appels récursifs, et toute la difficulté est de « dérécursiver » le premier appel.

15.1.6 Récursivité directe et croisée

Les procédures précédentes étaient bâties sur le même modèle. L'appel récursif est placé *directement* dans le corps de la procédure :

```
procédure P
    ... P ...
finproc {P}
```

Mais un sous-programme P peut faire appel à un ou plusieurs autres sous-programmes qui, en dernier lieu, fait appel au sous-programme P initial. On parle alors de récursivité *indirecte*. Lorsque seulement deux sous-programmes sont mis en jeu, on parle de récursivité *croisée* :

4. W. ACKERMANN, mathématicien allemand (1896-1962).

```
procédure P1
    ... P2 ...
finproc {P1}

procédure P2
    ... P1 ...
finproc {P2}
```

L'analyse des expressions arithmétiques en notation infixe est un bon exemple de récursivité indirecte. La grammaire donnée ci-dessous décrit des expressions formées d'opérateurs additifs et multiplicatifs, et d'opérandes. La grammaire fixe la priorité des opérateurs : les opérateurs additifs sont moins prioritaires que les opérateurs multiplicatifs.

```
<expression> = <terme>     | <expression> <op-add> <terme>
<terme>      = <facteur>   | <terme> <op-mult> <facteur>
<facteur>    = <variable> | <entier> | '(' <expression> ')'
```

Chaque entité, `<expression>`, `<terme>` ou `<facteur>` se décrit par une procédure. La récursivité indirecte a lieu dans `<facteur>` qui appelle récursivement `<expression>` pour analyser des expressions parenthésées.

La récursivité indirecte apparaît aussi dans les courbes de HILBERT[5]. Ces courbes sont telles que toute courbe H_i s'exprime en fonction d'une courbe H_{i-1}. Plus précisément, chaque courbe H_i consiste en l'utilisation de quatre parties (à l'échelle 1/2) de H_{i-1}. Ces quatre parties A, B, C et D sont reliées entre elles par trois segments de droite de la façon suivante :

```
A(i) : D(i-1) ← A(i-1) ↓ A(i-1) → B(i-1)
B(i) : C(i-1) ↑ B(i-1) → B(i-1) ↓ A(i-1)
C(i) : B(i-1) → C(i-1) ↑ C(i-1) ← D(i-1)
D(i) : A(i-1) ↓ D(i-1) ← D(i-1) ↑ C(i-1)
```

Ci-dessus, les flèches indiquent l'orientation des segments de droite qui relient les quatre courbes de niveau moins un. La figure 15.3 représente la superposition des courbes de niveau un à cinq.

Si l'on possède une fonction `tracer` qui trace un segment de droite dans le plan à partir de la position courante jusqu'au point de coordonnées (x,y), la procédure `A` s'écrit :

```
procédure A(donnée i: naturel)
{Rôle : tracer la partie A de la courbe de Hilbert de niveau i
        h longueur du segment qui relie les courbes de niveau i-1}
   si i>0 alors
       D(i-1)   x ← x-h   tracer(x,y)
       A(i-1)   y ← y-h   tracer(x,y)
       A(i-1)   x ← x+h   tracer(x,y)
       B(i-1)
   finsi
finproc {A}
```

5. DAVID HILBERT, mathématicien allemand (1862–1943), proposa cette courbe en 1890. Ce type de figure géométrique est plus connu aujourd'hui sous le nom de *fractale*.

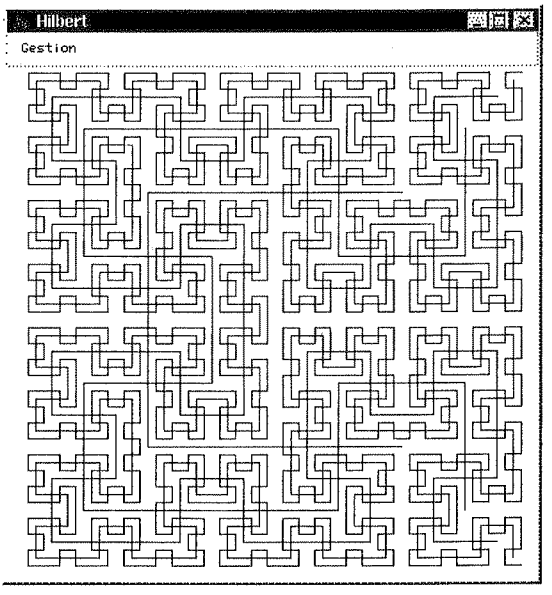


Figure 15.3 – *Courbes de Hilbert de niveau 5.*

L'écriture des procédures `B`, `C` et `D`, bâties sur ce modèle, est immédiate.

15.2 RÉCURSIVITÉ DES OBJETS

À l'instar de la récursivité des actions, un objet récursif est un objet qui contient un ou plusieurs composants du même type que lui. La récursivité des objets peut être également indirecte.

Imaginons que l'on veuille représenter la généalogie d'un individu. En plus de son identité, il est nécessaire de connaître son ascendance, c'est-à-dire l'arbre généalogique de sa mère et de son père. Il est clair que le type `Arbre Généalogique` que nous sommes amenés à définir est récursif :

```
classe Arbre Généalogique
        prénom type chaîne de caractères
        mère, père type Arbre Généalogique
finclasse {Arbre Généalogique}
```

Conceptuellement, une telle déclaration décrit une structure infinie, mais en général les langages de programmation ne permettent pas cette forme d'auto-inclusion des objets. Contrairement à la récursivité des actions, celle des objets ne crée pas « automatiquement » une infinité d'incarnations d'objets. C'est au programmeur de créer *explicitement* chacune de ces incarnations et de les relier entre elles.

La caractéristique fondamentale des objets récursifs est leur nature *dynamique*. Les objets que nous avons étudiés jusqu'à présent possédaient tous une taille fixe. La taille des objets récursifs pourra, quant à elle, varier au cours de l'exécution du programme.

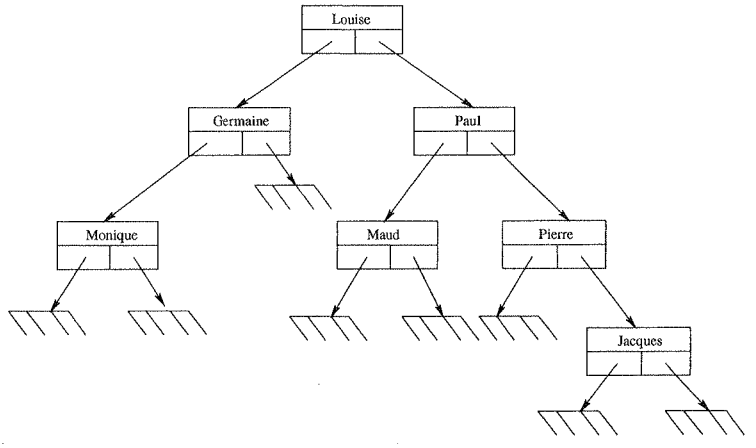


Figure 15.4 – *La généalogie de Louise.*

Pour mettre en œuvre la récursivité des objets, les langages de programmation proposent des outils qui permettent de créer dynamiquement un objet du type voulu et d'accéder à cet objet.

Des langages comme C ou PASCAL ne permettent pas une définition directement récursive des types, mais l'autorisent au moyen de *pointeurs*. Ils mettent à la disposition du programmeur des fonctions d'allocation mémoire, comme par exemple `malloc` et `new`, pour créer dynamiquement les incarnations des objets, auxquelles il accède par l'intermédiaire des pointeurs.

Pour de nombreuses raisons, mais en particulier pour des raisons de fiabilité de construction des programmes, les langages de programmation modernes ont abandonné la notion de pointeur. C'est le cas de JAVA qui permet des définitions d'objet vraiment récursives. Ainsi, le type `Arbre Généalogique` sera déclaré en JAVA comme suit :

```
class ArbreGénéalogique  {
        String prénom;
        // définition de l'ascendance
        ArbreGénéalogique mère, père;
        // le constructeur
        ArbreGénéalogique(String s) {
            prénom=s;
        }
} // ArbreGénéalogique
```

Cette définition est possible en JAVA car les attributs `mère` et `père` sont des *références* à des objets de type `ArbreGénéalogique` et *non pas* l'objet lui-même. L'arbre généalogique de *Louise*, donné par la figure 15.4, est produit par la séquence d'instructions suivante :

```
ArbreGénéalogique ag;
ag = new ArbreGénéalogique("Louise");
ag.mère = new ArbreGénéalogique("Germaine");
ag.père = new ArbreGénéalogique("Paul");
ag.mère.mère = new ArbreGénéalogique("Monique");
```

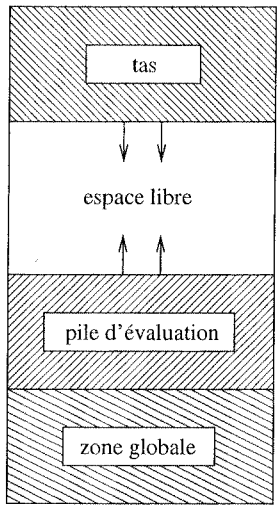


Figure 15.5 – *Organisation de la mémoire.*

```
ag.père.mère = new ArbreGénéalogique("Maud");
ag.père.père = new ArbreGénéalogique("Pierre");
ag.père.père.père = new ArbreGénéalogique("Jacques");
```

Chacun des ascendants, lorsqu'il est connu, est créé grâce à l'opérateur **new** et est relié à sa propre ascendance par l'intermédiaire des références. Celles-ci sont représentées sur la figure 15.4 par des flèches. Le symbole de la terre des électriciens indique la fin de la récursivité. En JAVA, il correspond à une référence égale à la constante **null**.

Les supports d'exécution des langages de programmation placent les objets dynamiques dans une zone spéciale, appelée *tas*. La figure 15.5 montre l'organisation classique de la mémoire lors de l'exécution d'un programme. La *zone globale* est une zone de taille fixe qui contient des constantes et des données globales du programme. La *pile d'évaluation*, dont la taille varie au cours de l'exécution du programme, contient l'empilement des zones locales des sous-programmes appelés avec leur pile d'évaluation des expressions. Enfin, *le tas* contient les objets alloués dynamiquement par le programme. Le tas croît en direction de la pile d'évaluation. S'ils se rencontrent, le programme s'arrête faute de place mémoire pour s'exécuter.

Selon les langages, la suppression des objets dynamiques, c'est-à-dire la libération de la place mémoire qu'ils occupent dans le tas, peut être à la charge du programmeur, ou laisser au support d'exécution. La suppression des objets dynamiques est une source de nombreuses erreurs, et il est préférable que le langage automatise la destruction des objets qui ne servent plus. C'est le choix fait par JAVA.

Dans les prochains chapitres, nous aurons souvent l'occasion de mettre en pratique des définitions d'objet récursif. De nombreuses structures de données que nous aborderons seront définies de façon récursive et les algorithmes qui les manipuleront seront eux-mêmes naturellement récursifs.

15.3 EXERCICES

Exercice 15.1. Programmez en JAVA les algorithmes `fibonacci` et `TroisdeHanoï` donnés à la page 159.

Exercice 15.2. En vous inspirant de la procédure `écrireChiffres`, écrivez de façon récursive et itérative la fonction `convertirRomain` qui retourne la représentation romaine d'un entier. On rappelle que les nombres romains sont découpés en quatre tranches : milliers, centaines, dizaines et unités (dans cet ordre). Dans chaque tranche, on écrit de zéro à quatre chiffres et jamais plus de trois chiffres identiques consécutifs. Les tranches nulles ne sont pas représentées. Les chiffres romains sont $I = 1$, $V = 5$, $X = 10$, $L = 50$, $C = 100$, $D = 500$, et $M = 1000$. Par exemple, $49 = XLIX$, $703 = DCCIII$ et $2000 = MM$.

Exercice 15.3. Écrivez une procédure qui engendre les $n!$ permutations de n éléments $a_1, \cdots, a_n$. La tâche consistant à engendrer les $n!$ permutations des éléments $a_1, \cdots, a_n$ peut être décomposée en n sous-tâches de génération de toutes les permutations de $a_1, \cdots, a_{n-1}$ suivies de a_n, avec échange de a_i et a_n dans la i^e sous-tâche.

Exercice 15.4. Écrivez un programme qui recopie sur la sortie standard un fichier de texte. Lorsqu'il reconnaît dans le fichier une directive de la forme `!`*nom-de-fichier*, il inclut le contenu du fichier en lieu et place de la directive. Évidemment, un fichier inclus peut contenir une ou plusieurs directives d'inclusion.

Exercice 15.5. Soit une fonction continue f définie sur un intervalle $[a,b]$. On cherche à trouver un zéro de f, c'est-à-dire un réel $x \in [a,b]$ tel que $f(x) = 0$. Si la fonction admet plusieurs zéros, n'importe lequel fera l'affaire. S'il n'y en a pas, il faudra le signaler.

Dans le cas où $f(a).f(b) < 0$, on est sûr de la présence d'un zéro. Lorsque $f(a).f(b) > 0$, il faut rechercher un sous-intervalle $[\alpha,\beta]$, tel que $f(\alpha).f(\beta) < 0$.

L'algorithme procède par dichotomie, c'est-à-dire qu'il va diviser l'intervalle de recherche en deux moitiés à chaque étape. Si l'un des deux nouveaux intervalles, par exemple $[\alpha,\beta]$, est tel que $f(\alpha).f(\beta) < 0$, on sait qu'il contient un zéro puisque la fonction est continue : on poursuivra alors la recherche dans cet intervalle.

En revanche, si les deux demi intervalles sont tels que f a le même signe aux deux extrémités, la solution, si elle existe, sera dans l'un ou l'autre de ces deux demi intervalles. Dans ce cas, on prendra arbitrairement l'un des deux demi intervalles pour continuer la recherche ; en cas d'échec on reprendra le deuxième demi intervalle qui avait été provisoirement négligé.

Écrivez de façon récursive l'algorithme de recherche d'un zéro, à ε près, de la fonction f.

Exercice 15.6. À partir de la petite grammaire d'expression donnée à la page 165, écrivez en JAVA un évaluateur d'expressions arithmétiques infixes. Pour simplifier, vous ne traiterez pas la notion de variable dans facteur.

Chapitre **16**

Structures de données

Le terme *structure de données* désigne une composition de données unies par une même sémantique. Mais, cette sémantique ne se réduit pas à celle des types (élémentaires ou structurés) des langages de programmation utilisés pour programmer la structure de données. Dès le début des années soixante-dix, C.A.R. HOARE [Hoa72] mettait en avant l'idée qu'une donnée représente avant tout une *abstraction* du monde réel définie en terme de structures abstraites, et qui n'est d'ailleurs pas nécessairement mise en œuvre à l'aide d'un langage de programmation particulier. D'une façon plus générale, un programme peut être lui-même modélisé en termes de données abstraites munies d'opérations abstraites.

Ces réflexions ont conduit à définir une structure de données comme une donnée abstraite, dont le comportement est modélisé par des opérations abstraites. C'est à partir du milieu des années soixante-dix que la théorie des *types abstraits algébriques* est apparue pour décrire les structures de données. Définis en termes de *signature*, les types abstraits doivent d'une part, garantir leur *indépendance* vis-à-vis de toute mise en œuvre particulière, et d'autre part, offrir un support de *preuve de la validité* de leurs opérations.

Dans ce chapitre, nous verrons comment spécifier une structure de données à l'aide d'un type abstrait, et comment l'implanter dans un langage de programmation particulier comme JAVA. Dans les chapitres suivants, nous présenterons plusieurs structures de données fondamentales, que tout informaticien doit connaître. Il s'agit des structures linéaires, de la structure de graphe, des structures arborescentes et des tables.

16.1 DÉFINITION D'UN TYPE ABSTRAIT

Un type abstrait est décrit par sa *signature* qui comprend :

- une déclaration des ensembles définis et utilisés ;
- une description fonctionnelle des opérations ;
- une description axiomatique de la sémantique des opérations.

Dans ce qui suit nous définirons (partiellement) les types abstraits *EntierNaturel* et *Ensemble*. Le premier décrit l'ensemble $\mathbb{N}$ des entiers naturels et le second des ensembles d'éléments quelconques.

➤ Déclaration des ensembles

Cette déclaration indique le nom du type abstrait à définir, ainsi que certaines constantes qui jouent un rôle particulier. La notation :

EntierNaturel. $0 \in$ *EntierNaturel*

déclare le type abstrait *EntierNaturel* des entiers naturels, qui possède un élément particulier dénoté 0.

La déclaration ensembliste de certains types abstraits nécessite de mentionner d'autres types abstraits. Ces derniers sont introduits par le mot-clé **utilise**. Le type abstrait *Ensemble* utilise deux autres types abstraits, *booléen* et $\mathcal{E}$. Il possède la définition suivante :

Ensemble **utilise** $\mathcal{E}$, booléen. $\emptyset \in$ *Ensemble*

où $\mathcal{E}$ définit les éléments d'un ensemble, et $\emptyset$ un ensemble vide. Remarquez que les types abstraits utilisés n'ont pas nécessairement besoin d'être au préalable entièrement définis. Pour la spécification du type abstrait *Ensemble* de ce chapitre, seule la connaissance des deux éléments vrai et faux de l'ensemble des booléens nous est utile.

➤ Description fonctionnelle

La définition fonctionnelle présente les signatures des opérations du type abstrait. Pour chaque opération, sa signature indique son nom et ses ensembles de départ et d'arrivée. L'ensemble des entiers naturels peut être décrit à l'aide de l'opération *succ* qui, pour un entier naturel, fournit son successeur. Il est également possible de définir les opérations arithmétiques $+$ et $\times$.

$$\begin{array}{lcll} \text{succ} & : & \textit{EntierNaturel} & \rightarrow \textit{EntierNaturel} \\ + & : & \textit{EntierNaturel} \times \textit{EntierNaturel} & \rightarrow \textit{EntierNaturel} \\ \times & : & \textit{EntierNaturel} \times \textit{EntierNaturel} & \rightarrow \textit{EntierNaturel} \end{array}$$

Si on munit le type abstrait *Ensemble* des opérations *est-vide?*, $\in$, *ajouter* et *union*, le type abstrait contiendra les signatures suivantes :

est-vide?	:	*Ensemble*	→	booléen
$\in$	:	*Ensemble* $\times \mathcal{E}$	→	booléen
ajouter	:	*Ensemble* $\times \mathcal{E}$	→	*Ensemble*
enlever	:	*Ensemble* $\times \mathcal{E}$	→	*Ensemble*
union	:	*Ensemble* $\times$ *Ensemble*	→	*Ensemble*

Description axiomatique

La définition axiomatique décrit la *sémantique* des opérations du type abstrait. Il est clair que les définitions ensembliste et fonctionnelle précédentes ne suffisent pas à exprimer ce qu'est le type *EntierNaturel* ou le type *Ensemble*. Le choix des noms des opérations nous éclaire, mais il existe par exemple une infinité de fonctions de $\mathbb{N}$ dans $\mathbb{N}$, et pour l'instant, rien ne distingue réellement la sémantique des opérations $+$ et $\times$.

Il faut donc spécifier de façon formelle les propriétés des opérations du type abstrait, ainsi que leur domaine de définition lorsqu'elles correspondent à des fonctions partielles, comme par exemple *enlever*. Pour cela, on utilise des *axiomes* qui mettent en jeu les ensembles et les opérations. Pour le type *EntierNaturel*, nous pouvons utiliser les axiomes proposés par G. PEANO [1] au siècle dernier.

(1) $\forall x \in \mathit{EntierNaturel}, \exists\, x',\ \mathrm{succ}(x) = x'$
(2) $\forall x,x' \in \mathit{EntierNaturel},\ x \neq x' \Rightarrow \mathrm{succ}(x) \neq \mathrm{succ}(x')$
(3) $\nexists\, x \in \mathit{EntierNaturel},\ \mathrm{succ}(x) = 0$
(4) $\forall x \in \mathit{EntierNaturel},\ x + 0 = x$
(5) $\forall x,y \in \mathit{EntierNaturel},\ x + \mathrm{succ}(y) = \mathrm{succ}(x + y)$
(6) $\forall x \in \mathit{EntierNaturel},\ x \times 0 = 0$
(7) $\forall x,y \in \mathit{EntierNaturel},\ x \times \mathrm{succ}(y) = x + x \times y$

Le premier axiome indique que tout entier naturel possède un successeur. Le second, que deux entiers naturels distincts possèdent deux successeurs distincts. Le troisième axiome précise que 0 n'est le successeur d'aucun entier naturel. Enfin, les quatre dernières spécifient les opérations $+$ et $\times$. Grâce à ces axiomes, il est démontré que tous les théorèmes de l'arithmétique de G. PEANO sont vrais pour les entiers naturels.

Les axiomes suivants décrivent les opérations du type abstrait *Ensemble* :

(1) est-vide?$(\emptyset)$ = vrai
(2) $\forall x \in \mathcal{E}, \forall e \in \mathit{Ensemble}$, est-vide?$(\mathit{ajouter}(e,x))$ = faux
(3) $\forall x \in \mathcal{E}, x \in \emptyset$ = faux
(4) $\forall x,y \in \mathcal{E}, \forall e \in \mathit{Ensemble}$,
$x = y \Rightarrow y \in \mathrm{ajouter}(e,x)$ = vrai
$x \neq y \Rightarrow y \in \mathrm{ajouter}(e,x) = y \in e$
(5) $\forall x,y \in \mathcal{E}, \forall e \in$ Ensemble,
$x = y \Rightarrow y \in \mathrm{enlever}(e,x)$ = faux
$x \neq y \Rightarrow y \in \mathrm{enlever}(e,x) = y \in e$

1. GIUSEPPE PEANO, mathématicien italien (1858-1932).

(6) $\forall x \in \mathcal{E}, \forall e \in \textit{Ensemble}$,
$x \notin e \Rightarrow \nexists e' \in \textit{Ensemble}, e' = \text{enlever}(e,x)$
(7) $x \in \text{union}(e,e') \Rightarrow x \in e$ ou $x \in e'$

Notez que l'axiome (5) impose la présence dans l'ensemble de l'élément à retirer. La fonction *enlever* est une fonction partielle, et cet axiome en précise le domaine de définition.

Une des principales difficultés de la définition axiomatique est de s'assurer, d'une part, de sa *consistance*, c'est-à-dire qu'il n'y a pas d'axiomes qui se contredisent, et d'autre part, de sa *complétude*, c'est-à-dire que les axiomes définissent entièrement le type abstrait. [FGS90] distingue deux types d'opérations : les opérations *internes*, qui rendent un résultat de l'ensemble défini, et les *observateurs*, qui rendent un résultat de l'ensemble prédéfini, et propose de tester la complétude d'un type abstrait en vérifiant si l'on peut déduire de ses axiomes le résultat de chaque observateur sur son domaine de définition. Pour garantir la consistance, il suffit alors de s'assurer que chacune de ces valeurs est unique.

16.2 L'IMPLANTATION D'UN TYPE ABSTRAIT

L'*implantation* est la façon dont le type abstrait est programmé dans un langage particulier. Il est évident que l'implantation doit respecter la définition formelle du type abstrait pour être valide. Certains langages de programmation, comme ALPHARD ou EIFFEL, incluent des outils qui permettent de spécifier et de vérifier automatiquement les axiomes, c'est-à-dire de contrôler si les opérations du type abstrait respectent, au cours de leur utilisation, ses propriétés algébriques.

L'implantation consiste donc à choisir les structures de données *concrètes*, c'est-à-dire des types du langage d'écriture pour représenter les ensembles définis par le type abstrait, et de rédiger le corps des différentes fonctions qui manipuleront ces types. D'une façon générale, les opérations des types abstraits correspondent à des sous-programmes de petite taille qui seront donc faciles à mettre au point et à maintenir.

Pour un type abstrait donné, plusieurs implantations possibles peuvent être développées. Le choix d'implantation du type abstrait variera selon l'utilisation qui en est faite et aura une influence sur la complexité des opérations.

Le concept de classe des langages à objets facilite la programmation des types abstraits dans la mesure où chaque objet porte ses propres données et les opérations qui les manipulent. Notez toutefois que les opérations d'un type abstrait sont associées à l'ensemble, alors qu'elles le sont à l'objet dans le modèle de programmation à objets. La majorité des langages à objets permet même de conserver la distinction entre la définition abstraite du type et son implantation grâce aux notions de *classe abstraite* ou d'*interface*.

En JAVA, l'interface suivante représente la définition fonctionnelle du type abstrait *Entier Naturel* :

```
public interface EntierNaturel {
    public EntierNaturel succ();
    public EntierNaturel plus(EntierNaturel n);
    public EntierNaturel mult(EntierNaturel n);
}
```

Dans la mesure où le langage JAVA n'autorise pas la surcharge des opérateurs, les symboles + et * n'ont pu être utilisés et les opérations d'addition et de multiplication ont été nommées.

La définition fonctionnelle du type abstrait *Ensemble* correspondra à la déclaration de l'interface suivante :

```
public interface Ensemble {
    public boolean estVide();
    public boolean dans(Object x);
    public void ajouter(Object x);
    public void enlever(Object x) throws ElémentAbsentException;
    public Ensemble union(Ensemble x);
}
```

Notez que la méthode `enlever` émet une exception si l'élément `x` à retirer n'est pas présent dans l'ensemble. L'exception traduit l'axiome (5) du type abstrait. D'une façon générale, les exceptions serviront à la définition des fonctions partielles des types abstraits.

La programmation des opérations dépendra des types de données choisis pour implanter le type abstrait. Pour un ensemble, il est possible de choisir un arbre ou une liste, eux-mêmes implantés à l'aide de tableau ou d'éléments chaînés. L'implantation de l'interface `Ensemble` aura par exemple la forme suivante :

```
public class EnsembleListeChaînée implements Ensemble {
    public EnsembleListeChaînée()
    {
        // Le constructeur
    }
    public boolean estVide()
    {
        ...
    }
    public boolean dans(Object x)
    {
        ...
    }
    public void ajouter(Object x)
    {
        ...
    }
    public void enlever(Object x) throws ElémentAbsentException
    {
        ...
    }
    ...
} // fin classe EnsembleListeChaînée
```

Dans cette partie implantation, le programmeur devra bien prendre soin d'interdire l'accès aux données concrètes, et de rendre publiques les opérations du type abstrait.

16.3 UTILISATION DU TYPE ABSTRAIT

Puisque la définition d'un type abstrait est indépendante de toute implantation particulière, l'utilisation du type abstrait devra se faire *exclusivement* par l'intermédiaire des opérations qui lui sont associées et en aucun cas en tenant compte de son implantation. D'ailleurs certains langages de programmation peuvent vous l'imposer, mais ce n'est malheureusement pas le cas de tous les langages de programmation et c'est alors au programmeur de faire preuve de rigueur !

Les en-têtes des fonctions et des procédures du type abstrait et les affirmations qui définissent leur rôle représentent l'interface entre l'utilisateur et le type abstrait. Ceci permet évidemment de manipuler le type abstrait sans même que son implantation soit définie, mais aussi de rendre son utilisation *indépendante* vis-à-vis de tout changement d'implantation. Par exemple, l'écriture de l'énoncé conditionnel donné ci-dessous sera valide quelle que soit l'implantation choisie pour le type `Ensemble`.

```
void uneMéthode(Ensemble e) {
  ...
  if (e.dans(x)) {
      ...
  }
  ...
}
```

Enfin, on peut ajouter que si le type abstrait est juste et validé, il y a plus de chances que son utilisation à l'aide de ses fonctions soit elle aussi juste.

16.4 GÉNÉRICITÉ

La définition du type abstrait *Ensemble* n'impose aucune restriction sur la nature des éléments des ensembles. Ceux-ci peuvent être différents ou semblables. Les opérations d'appartenance ou d'union doivent s'appliquer aussi bien à des ensembles d'entiers qu'à des ensembles de machines à laver, ou encore des ensembles d'ensembles.

L'implantation du type abstrait doit être alors *générique*, c'est-à-dire qu'elle doit permettre de manipuler des éléments de n'importe quel type. Souvent, il est même souhaitable d'imposer que tous les éléments soient d'un type donné. De nombreux langages de programmation (ADA, C++, EIFFEL, *etc.*) incluent dans leur définition la notion de généricité et proposent des mécanismes de construction de types génériques. Ils offrent la définition de types génériques auxquels on passe en paramètre le type désiré. Dans les déclarations suivantes, `e1` est un ensemble d'entiers, `e2` un ensemble de chaînes de caractères, et `e3` un ensemble d'ensembles d'entiers naturels.

```
variable e1 type Ensemble(entier)
variable e1 type Ensemble(chaîne de caractères)
variable e2 type Ensemble(Ensemble(EntierNaturel))
```

Dans sa version actuelle, le langage JAVA permet une généricité restreinte grâce au polymorphisme et au type `Object`. Par exemple, un élément d'un `Ensemble` peut être un

entier alors qu'un autre peut être une chaîne de caractères, mais rien dans le langage ne peut contraindre les éléments d'un ensemble à être tous du même type. Des propositions, comme par exemple [BOSW98], sont en cours d'étude pour inclure cette possibilité dans JAVA et permettre un contrôle de type à la compilation. Dans l'état actuel du langage, il est toutefois possible de faire cette vérification dynamiquement à l'exécution. Pour cela, un `Ensemble` doit mémoriser le type de ses éléments. Dans la classe ci-dessous, ce type est conservé dans l'attribut `typeDesÉléments` à la construction d'un ensemble. La méthode `ajouter` peut alors vérifier que l'élément que l'on cherche à ajouter est bien du type des éléments de l'ensemble.

```
public class EnsembleListeChaînée implements Ensemble {
     protected Class typeDesÉléments;
     public EnsembleListeChaînée(Class c) {
             ...
             typeDesÉléments = c;
             ...
     }

     public boolean ajouter(Object e) {
         if (e.getClass() != typeDesÉléments)
              throw new TypeIncompatibleException();
         ...
     }
     ...
} // fin classe EnsembleListeChaînée
```

À la création d'un objet de type `Ensemble`, le type des éléments est fourni grâce à la méthode `forName` ou en suffixant le paramètre par `.class`. Les déclarations suivantes construisent toutes les deux un ensemble d'entiers.

```
Ensemble E =
            new EnsembleListeChaînée(Class.forName("java.lang.Integer"));
```

ou

```
Ensemble E = new EnsembleListeChaînée(java.lang.Integer.class);
```

Chapitre **17**

Structures linéaires

Les structures linéaires sont un des modèles de données les plus élémentaires et utilisés dans les programmes informatiques. Elles organisent les données sous forme de *séquence* non ordonnée d'éléments accessibles de façon séquentielle. Tout élément d'une séquence, sauf le dernier, possède un *successeur*. Une séquence s constituée de n éléments sera dénotée comme suit :

$$s = < e_1\ e_2\ e_3\ \ldots\ e_n >$$

et la séquence vide :

$$s = <>$$

Les opérations d'ajout et de suppression d'éléments sont les opérations de base des structures linéaires. Selon la façon dont procèdent ces opérations, nous distinguerons différentes sortes de structures linéaires. Les *listes* autorisent des ajouts et des suppressions d'éléments n'importe où dans la séquence, alors que les *piles*, les *files* et les *dèques* ne les permettent qu'aux extrémités. On considère que les piles, les files et les dèques sont des formes particulières de liste linéaire. Dans ce chapitre, nous commencerons par présenter la forme générale, puis nous étudierons les trois formes particulières de liste.

17.1 LES LISTES

La *liste* définit une forme générale de séquence. Une liste est une séquence finie d'éléments repérés selon leur *rang*. S'il n'y a pas de relation d'ordre sur l'ensemble des éléments de la séquence, il en existe une sur le rang. Le rang du premier élément est 1, le rang du second est

2, et ainsi de suite. L'ajout et la suppression d'un élément peut se faire à n'importe quel rang *valide* de la liste.

17.1.1 Définition abstraite

➢ Ensembles

$\mathcal{L}iste$ est l'ensemble des listes linéaires non ordonnées dont les éléments appartiennent à un ensemble $\mathcal{E}$ quelconque. L'ensemble des entiers représente le rang des éléments. La constante *listevide* est la liste vide.

$\mathcal{L}iste$ **utilise** $\mathcal{E}$, naturel et entier
listevide $\in \mathcal{L}iste$

➢ Description fonctionnelle

Le type abstrait $\mathcal{L}iste$ définit les quatre opérations de base suivantes :

longueur	:	$\mathcal{L}iste$	$\rightarrow$	*naturel*
ième	:	$\mathcal{L}iste \times entier$	$\rightarrow$	$\mathcal{E}$
supprimer	:	$\mathcal{L}iste \times entier$	$\rightarrow$	$\mathcal{L}iste$
ajouter	:	$\mathcal{L}iste \times entier \times \mathcal{E}$	$\rightarrow$	$\mathcal{L}iste$

L'opération *longueur* retourne le nombre d'éléments de la liste. L'opération *ième* retourne l'élément d'un rang donné. Enfin, *supprimer* (resp. *ajouter*) supprime (resp. ajoute) un élément à un rang donné.

➢ Description axiomatique

Les axiomes suivants décrivent les quatre opérations applicables sur les listes. La *longueur* d'une liste vide est égale à zéro. L'ajout d'un élément dans la liste augmente sa longueur de un, et sa suppression la réduit de un.

$\forall l \in \mathcal{L}iste$, et $\forall e \in \mathcal{E}$
(1) longueur(listevide) $= 0$
(2) $\forall r,\ 1 \leqslant r \leqslant$ longueur(l), longueur(supprimer(l,r)) $=$ longueur$(l) - 1$
(3) $\forall r,\ 1 \leqslant r \leqslant$ *longueur*$(l) + 1$, longueur(ajouter(l,r,e)) $=$ longueur$(l) + 1$

L'opération *ième* renvoie l'élément de rang r, et n'est définie que si le rang est valide.

(4) $\forall r,\ r < 1$ et $r >$ longueur(l), $\nexists e,\ e =$ ième(l,r)

L'opération *supprimer* retire un élément qui appartient à la liste, c'est-à-dire dont le rang est compris entre un et la longueur de la liste. Les axiomes suivants indiquent que le rang des éléments à droite de l'élément supprimé est décrémenté de un.

(5) $\forall r,\ 1 \leqslant r \leqslant$ longueur(l) et $1 \leqslant i < r$ ième(supprimer$(l,r),i)$ $=$ ième(l,i)

(6) $\forall r, 1 \leqslant r \leqslant \text{longueur}(l) \text{ et } r \leqslant i \leqslant \text{longueur}(l) - 1$,
ième(supprimer(l,r),i) = ième(l,$i+1$)
(7) $\forall r, r < 1 \text{ et } r > \text{longueur}(l), \nexists l', l' = \text{supprimer}(l,r)$

L'opération *ajouter* insère un élément à un rang compris entre un et la longueur de la liste plus un. Le rang des éléments à la droite du rang d'insertion est incrémenté de un.

(8) $\forall r, 1 \leqslant r \leqslant \text{longueur}(l) + 1 \text{ et } 1 \leqslant i < r$,
ième(ajouter(l,r,e),i) = ième(l,i)
(9) $\forall r, 1 \leqslant r \leqslant \text{longueur}(l) + 1 \text{ et } r = i$, ième($ajouter(l,r,e),i$) = e
(10) $\forall r, 1 \leqslant r \leqslant \text{longueur}(l) + 1 \text{ et } r < i \leqslant \text{longueur}(l) + 1$,
ième(ajouter(l,r,e),i) = ième(l,$i-1$)
(11) $\forall r, r < 1 \text{ et } r > \text{longueur}(l) + 1, \nexists l', l' = \text{ajouter}(l,r,e)$

17.1.2 L'implantation en Java

La description fonctionnelle du type abstrait *Liste* est traduite en JAVA par l'interface suivante :

```
public interface Liste {
  public int longueur();
  public Object ième(int r) throws RangInvalideException;
  public void supprimer(int r) throws RangInvalideException;
  public void ajouter(int r, Object e) throws RangInvalideException;
}
```

Les éléments de la liste sont déclarés de type `Object` permettant ainsi de définir une liste d'éléments de type quelconque. Les méthodes d'accès aux éléments de la liste peuvent lever l'exception `RangInvalideException` si elles tentent d'accéder à un rang invalide. Cette exception est simplement définie par la déclaration :

```
public class RangInvalideException extends RuntimeException {
    public RangInvalideException() {
        super();
    }
}
```

Afin d'insister sur l'indépendance du type abstrait vis-à-vis de son implantation, nous présenterons successivement deux sortes d'implantations. La première utilise des tableaux et la seconde des structures chaînées. Les tableaux offrent une représentation contiguë des éléments, et permettent un accès direct aux éléments qui les composent, mais ont comme principal inconvénient de fixer la taille de la structure de données. Par exemple, si pour représenter une liste, on déclare un tableau de cent composants alors quelle que soit la longueur effective de la liste, l'encombrement mémoire utilisé sera celui des cent éléments. Au contraire, les structures chaînées sont des structures dynamiques qui permettent d'adapter la taille de la structure de données au nombre effectif d'éléments. L'espace mémoire nécessaire pour mémoriser un élément est plus important, et le temps d'accès aux éléments est en général plus coûteux parce qu'il a lieu de façon indirecte.

Utilisation d'un tableau

La méthode qui vient en premier à l'esprit, lorsqu'on mémorise les éléments d'une liste dans un tableau, est de conserver systématiquement le premier élément à la première place du tableau, et de ne faire varier qu'un indice de fin de liste. La figure 17.1 montre une séquence de cinq entiers placée dans un tableau nommé `éléments`. L'attribut `lg`, qui indique la longueur de la liste, donne également l'indice de fin de liste.

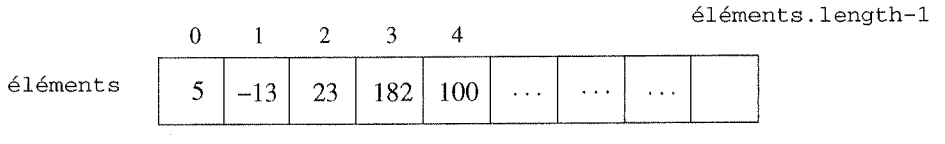


Figure 17.1 – *Une liste dans un tableau.*

L'algorithme de l'opération *ième* est très simple, puisque le tableau permet un accès *direct* à l'élément de rang `r`. La complexité de cet algorithme est donc $\mathcal{O}(1)$. Notez que pour accéder à un élément de la liste l'antécédent de l'opération doit être vérifié.

```
Algorithme ième(r)
    {Antécédent : 1 ⩽ r ⩽ longueur(l)}
    rendre éléments[r-1] {les valeurs débutent à l'indice 0}
```

L'opération de suppression d'un élément de la liste provoque un décalage des éléments qui se situent à droite du rang de suppression. Pour une liste de n éléments, la complexité de cette opération est $\mathcal{O}(n)$, et l'algorithme qui la décrit est le suivant :

```
Algorithme supprimer(r)
    {Antécédent : 1 ⩽ r ⩽ longueur(l)}
    pourtout i de r à lg faire
        éléments[i-1] ← éléments[i]
    finpour
    lg ← lg-1
```

L'opération d'ajout d'un élément `e` au rang `r` consiste à décaler d'une position vers la droite tous les éléments à partir du rang `r`. Le nouvel élément est inséré au rang `r`. Dans la plupart des langages de programmation, une déclaration de tableau fixe sa taille à la compilation. Quel que soit le constructeur choisi, un objet, instance de la classe `ListeTableau`, possédera un nombre d'éléments fixe. Ceci contraint la méthode `ajouter` à vérifier si le tableau `éléments` dispose d'une place libre avant d'ajouter un nouvel élément. Comme pour l'opération de suppression, la complexité de cet algorithme est $\mathcal{O}(n)$. L'algorithme est le suivant :

```
Algorithme ajouter(r, e)
    {Antécédent : 1 ⩽ r ⩽ longueur(l)+1
                  et longueur(l)+1<éléments.length}
```

```
    pourtout i de lg-1 à r-1 faire
            éléments[i+1] ← éléments[i]
    finpour
    {r-1 est l'indice d'insertion}
    éléments[r-1] ← e
    lg ← lg+1
```

Remarquez que l'antécédent spécifie que que le tableau doit posséder une place libre pour l'élément à ajouter. S'il n'est pas vérifié, cela se traduira en JAVA par l'exception `ListePleineException`. Celle-ci n'est pas définie par le type abstrait, mais est induite par le choix des données concrètes pour son implantation. La définition de l'exception `ListePleineException` est semblable à celle de `ListeVideException` :

```
public class ListePleineException extends RuntimeException {
    public ListePleineException() {
        super();
    }
}
```

La classe `ListeTableau` suivante donne l'implantation complète d'une liste à l'aide d'un tableau géré selon les algorithmes précédents :

```
public class ListeTableau implements Liste {
    protected static final int MAXÉLÉM=100;
    protected int lg;
    protected Object [] éléments;
    protected Class typeDesÉléments;
    public ListeTableau(Class c) { this(MAXÉLÉM,c); }
    public ListeTableau(int n, Class c) {
        éléments=new Object[n];
        typeDesÉléments=c
    }
    public int longueur() {
        return lg;
    }
    public Object ième(int r) throws RangInvalideException {
        if (r<1 || r>lg)
            throw new RangInvalideException();
        return éléments[r-1];
    }
    public void supprimer(int r) throws RangInvalideException {
        if (r<1 || r>lg)
            throw new RangInvalideException()
        // décaler les éléments vers la gauche;
        for (int i=r; i<lg; i++)
            éléments[i-1]=éléments[i];
        lg--;
    }
```

```
    public void ajouter(int r, Object e) throws RangInvalideException
    {
        if (e.getClass()!=typeDesÉléments)
            throw new TypeIncompatibleException();
        if (lg==éléments.length)
            throw new ListePleineException();
        if (r<1 || r>lg+1)
            throw new RangInvalideException();
        // décaler les éléments vers la droite
        for (int i=lg; i>=r; i--)
            éléments[i]=éléments[i-1];
        // r-1 est l'indice de l'élément à ajouter
        éléments[r-1]=e;
        lg++;
    }
} // fin classe ListeTableau
```

On peut remarquer que la suppression de l'élément de tête est très coûteuse puisqu'elle provoque un décalage de tous les éléments de la liste. Il possible d'éviter cela en gérant le tableau de façon *circulaire*.

La gestion circulaire du tableau se fait à l'aide de deux indices : un indice de tête qui désigne le premier élément de la liste, et un indice de queue qui indique l'emplacement libre après le dernier élément de la liste. Pour un tableau du langage JAVA, ces deux indices ont pour valeur initiale 0, et l'indice du dernier composant est `length-1`.

Les indices de tête ou de queue sont incrémentés ou décrémentés de un à chaque ajout ou suppression. Lorsqu'on incrémente un indice en fin de tableau, sa prochaine valeur est alors l'indice du premier composant du tableau :

```
tête = tête==éléments.length-1 ? 0 : ++tête;
queue = queue==éléments.length-1 ? 0 : ++queue;
```

De même, lorsqu'on décrémente un indice en début de tableau, sa prochaine valeur est alors l'indice du dernier composant du tableau :

```
tête = tête==0 ? éléments.length-1 : --tête;
queue = queue==0 ? éléments.length-1 : --queue;
```

Notez que l'indice de tête ne précède pas nécessairement l'indice de queue, et qu'au gré des ajouts et des suppressions l'indice de tête peut être aussi bien inférieur que supérieur à l'indice de queue. La figure 17.2 donne deux dispositions possibles de la séquence < 5 20 4 9 45 > dans le tableau `éléments`.

Dans le cas général de la suppression ou de l'ajout d'un élément qui n'est pas situé à l'une des extrémités de la liste, le décalage d'une partie des éléments est nécessaire comme dans la méthode de gestion du tableau précédente. Ce décalage est lui-même circulaire et se fait modulo la taille du tableau. L'indice d'un élément de rang `r` est égal à `tête+r-1`.

Nous définissons la classe `ListeTableauCirculaire` en remplaçant les méthodes `ième`, `supprimer` et `ajouter` par celles données ci-dessous :

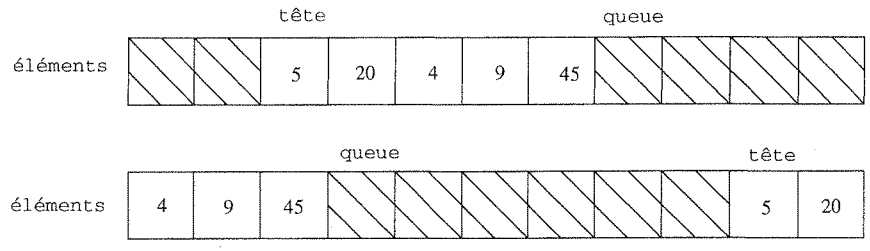


Figure 17.2 – *Gestion circulaire d'un tableau.*

```
public Object ième(int r) throws RangInvalideException
{
  if (r<1 || r>lg) throw new RangInvalideException();
  return éléments[(tête+r-1) % éléments.length];
}
public void supprimer(int r) throws RangInvalideException
{
  if (r<1 || r>lg) throw new RangInvalideException();
  if (r==lg) // supprimer le dernier élément
      queue = queue==0 ? éléments.length-1 : --queue;
  else
      if (r==1)  // supprimer le premier élément
          tête = tête==éléments.length-1 ? 0 : ++tête;
      else { // décaler les éléments
          for (int i=tête+r; i<lg+tête; i++)
              // i>0
              éléments[(i-1) % éléments.length] =
                                  éléments[i % éléments.length];
          queue = queue==0 ? éléments.length-1 : --queue;
      }
  lg--;
}
public void ajouter(int r, Object e) throws RangInvalideException
{
  if (e.getClass()!=typeDesÉléments)
            throw new TypeIncompatibleException();
  if (lg==éléments.length)
       throw new ListePleineException();
  if (r<1 || r>lg +1)
       throw new RangInvalideException();
  if (r==lg+1) { // ajouter en queue
      éléments[queue]=e;
      queue = queue==éléments.length-1 ? 0  : ++queue;
  } else
      if (r==1)  { // ajouter en tête
          tête = tête==0 ? éléments.length-1 : --tête;
          éléments[tête]=e;
      }
```

```
        else { // décaler les éléments
            for (int i=lg+tête; i >= r+tête; i--)
                // i > 0
                éléments[i % éléments.length ]=
                        éléments[(i-1) % éléments.length];
            // tête+r-1 est l'indice d'insertion
            éléments[tête+r-1]=e;
            queue = queue==éléments.length-1 ? 0 : ++queue;
        }
    lg++;
}
```

Utilisation d'une structure chaînée

Une structure chaînée est une structure dynamique formée de *nœuds* reliés par des *liens*. Les figures 17.3 et 17.4 montrent les deux types de structures chaînées que nous utiliserons pour représenter une liste. Dans cette figure, les nœuds sont représentés par des boîtes rectangulaires, et les liens par des flèches.

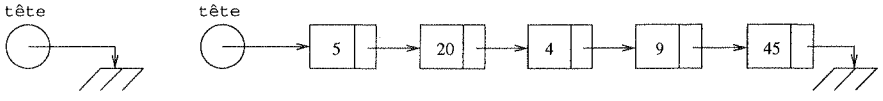


Figure 17.3 – *Liste avec chaînage simple.*

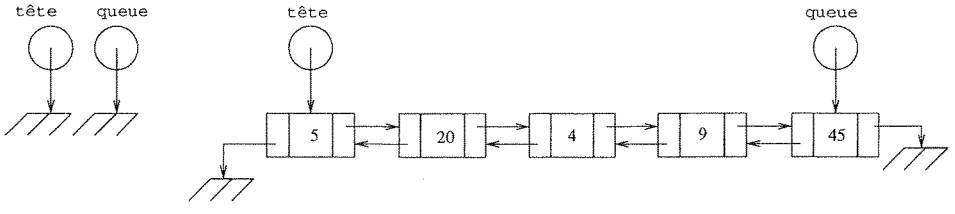


Figure 17.4 – *Liste avec chaînage double.*

Avec la première organisation, chaque élément est relié à son successeur par un simple lien et l'accès se fait de la gauche vers la droite à partir de la tête de liste qui est une référence sur le premier nœud. Si sa valeur est égale à **`null`**, la liste est considérée comme vide.

Dans la seconde organisation, chaque élément est relié à son prédécesseur et à son successeur, permettant un parcours de la liste dans les deux sens depuis la tête ou la queue. La tête est une référence sur le premier nœud et la queue sur le dernier. La liste est vide lorsque la tête et la queue sont toutes deux égales à la valeur **`null`**.

Nous représenterons un lien par la classe `Lien` qui définit simplement un attribut `suivant` de type `Lien` pour désigner le nœud suivant. Cette classe possède une méthode qui renvoie la valeur de cet attribut et une autre qui la modifie. Le constructeur par défaut initialise l'attribut `suivant` à la valeur **`null`**.

```
public class Lien {
  protected Lien suivant;
  protected Lien suivant() { return suivant; }
  protected void suivant(Lien s) { suivant=s; }
} // fin classe Lien
```

Les *nœuds* sont représentés par la classe `Noeud` qui hérite de la classe `Lien` et l'étend par l'ajout de l'attribut `valeur` qui désigne la valeur du nœud, *i.e.* la valeur d'un élément de la séquence.

Cette classe possède deux constructeurs qui initialisent un nœud avec la valeur d'un élément particulier, et avec celle d'un lien sur un autre nœud. La méthode `valeur` renvoie la valeur du nœud, alors que la méthode `changerValeur` change sa valeur. La méthode `noeudSuivant` retourne ou modifie le nœud suivant.

```
public class Noeud extends Lien {
  protected Object valeur;
  public Noeud(Object e) { valeur=e; }
  public Noeud(Object e, Noeud s) { valeur=e; suivant(s); }
  public Object valeur() { return valeur; }
  public void changerValeur(Object e) { valeur=e; }
  public Noeud noeudSuivant() { return (Noeud) suivant(); }
  public void noeudSuivant(Noeud s) { suivant(s); }
}
```

Avec cette structure chaînée, les opérations `ième`, `supprimer`, et `ajouter` nécessitent toutes un parcours séquentiel de la liste et possèdent donc une complexité égale à $\mathcal{O}(n)$. On atteint le nœud de rang `r` en appliquant `r-1` fois l'opération `noeudSuivant` à partir de la tête de liste. Nous noterons ce nœud $\texttt{noeudSuivant}^{r-1}\texttt{(tête)}$. L'algorithme de l'opération `ième` s'écrit :

Algorithme `ième(r)`
{Antécédent : $1 \leqslant r \leqslant$ longueur(l)}
rendre $\texttt{noeudSuivant}^{r-1}\texttt{(tête).valeur()}$

Comme le montre la figure 17.5, la suppression d'un élément de rang `r` consiste à affecter au lien qui le désigne la valeur de son lien suivant. La flèche en pointillé représente le lien avant la suppression. Notez que si `r` est égal à un, il faut modifier la tête de liste.

Algorithme `supprimer(r)`
{Antécédent : $1 \leqslant r \leqslant$ longueur(l)}
si `r=1` **alors**
  `tête` $\leftarrow$ `noeudSuivant(tête)`
sinon
  $\texttt{noeudSuivant}^{r-2}\texttt{(tête).suivant} \leftarrow \texttt{noeudSuivant}^{r}\texttt{(tête)}$
finsi
`lg` $\leftarrow$ `lg-1`

L'ajout d'un élément `e` au rang `r` consiste à créer un nouveau nœud `n` initialisé à la valeur `e`, puis à relier le nœud de rang `r-1` à `n`, et enfin à relier le nœud `n` au nœud de rang `r`. Si

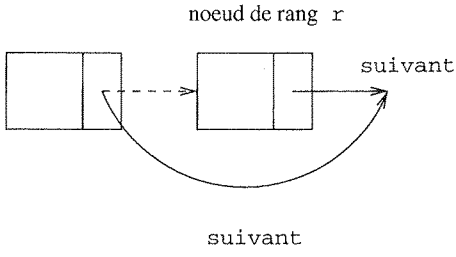


Figure 17.5 – *Suppression du nœud de rang r.*

l'élément est ajouté en queue de liste, son suivant est la valeur **null**. Comme précédemment, si `r=1`, il faut modifier la tête de liste.

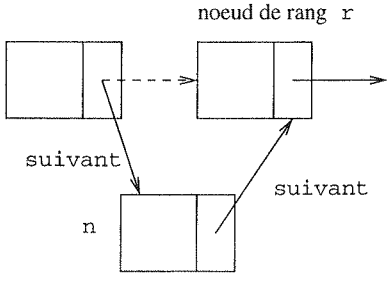


Figure 17.6 – *Ajout d'un nœud au rang r.*

Algorithme `ajouter(r, e)`

{Antécédent : 1 ⩽ r ⩽ longueur(l)+1}

`n ← créer Noeud(e)`

`noeudSuivant`$^{r-2}$`(tête).suivant ←n`

`n.suivant ← noeudSuivant`$^{r-1}$`(tête)`

`lg ← lg+1`

Nous pouvons donner l'écriture complète de la classe `ListeChaînée`.

```
public class ListeChaînée implements Liste {
    protected int lg;
    protected Noeud tête;
    protected Class typeDesÉléments;
    public ListeChaînée(Class c) { typeDesÉléments=c; }
    public int longueur() { return lg; }
    public Object ième(int r) throws RangInvalideException {
        if (r<1 || r>lg)
            throw new RangInvalideException();
        Noeud n=tête;
        for (int i=1; i<r; i++) n=n.noeudSuivant();
```

```
        // n désigne le nœud de rang r
        return n.valeur();
    }
    public void supprimer(int r) throws RangInvalideException {
        if (r<1 || r>lg)
             throw new RangInvalideException();
        if (r==1) // suppression en tête de liste
            tête=tête.noeudSuivant();
        else { // cas général, r>1
            Noeud p=null, q=tête;
            for (int i=1; i<r; i++) {
                p=q;
                q=q.noeudSuivant();
            }
            // q désigne l'élément de rang r et p son prédécesseur
            p.suivant(q.suivant());
        }
        lg--;
    }
    public void ajouter(int r, Object e) throws RangInvalideException
    {
        if (e.getClass()!=typeDesÉléments)
             throw new TypeIncompatibleException();
        if (r<1 || r>lg +1)
             throw new RangInvalideException();
        Noeud n=new Noeud(e);
        if (r==1) { // insertion en tête de liste
            n.suivant(tête);
            tête=n;
        }
        else { // cas général, r>1
            Noeud p=null, q=tête;
            for (int i=1; i<r; i++) {
                p=q;
                q=q.noeudSuivant();
            }
            // q désigne l'élément de rang r et p son prédécesseur
            p.suivant(n);
            n.suivant(q);
        }
        lg++;
    }
} // fin classe ListeChaînée
```

La structure qui chaîne les nœuds avec un simple lien ne permet qu'un parcours unidirectionnel de la liste. Il est pourtant utile dans certains cas d'autoriser un accès bidirectionnel, en particulier pour supprimer le dernier l'élément de la structure.

Le double chaînage est défini par la classe `LienDouble` qui étend la classe `Lien` en lui ajoutant un second lien, l'attribut `précédent`, dont la définition est semblable à l'attribut `suivant`. La déclaration de cette classe est la suivante :

```
public class LienDouble extends Lien {
    protected Lien précédent;
    protected Lien précédent() { return précédent; }
    protected void précédent(Lien s) { précédent=s; }
} // fin classe LienDouble
```

Chaque nœud d'une structure doublement chaînée est représenté par la classe `Noeud2` suivante qui étend la classe `LienDouble`.

```
public class Noeud2 extends LienDouble {
   protected Object valeur;
   public Noeud2(Object e) { valeur=e; }
   public Noeud2(Object e, Noeud2 p, Noeud2 s) {
         valeur=e; précédent(p); suivant(s);
   }
   public Object valeur() { return valeur; }
   public void changerValeur(Object e) { valeur=e; }
   public Noeud2 noeud2Suivant() { return (Noeud2) suivant(); }
   public void  noeudSuivant(Noeud2 s) { suivant(s); }
   public Noeud2 noeud2Précédent() {return (Noeud2) précédent(); }
   public void  noeudPrécédent(Noeud2 s) { précédent(s); }
}
```

Une liste doublement chaînée possède une tête qui désigne le premier élément de la liste, et une queue qui indique le dernier élément.

L'opération `ième` est identique à celle qui utilise une liste simplement chaînée. Sa complexité est $\mathcal{O}(n)$.

La suppression d'un élément de rang `r` nécessite de mettre à jour le lien `précédent` du nœud de rang `r+1` s'il existe (voir la figure 17.7). La complexité est $\mathcal{O}(n)$. Notez que la suppression du dernier élément de la liste est une simple mise à jour de l'attribut `queue` et sa complexité est $\mathcal{O}(1)$.

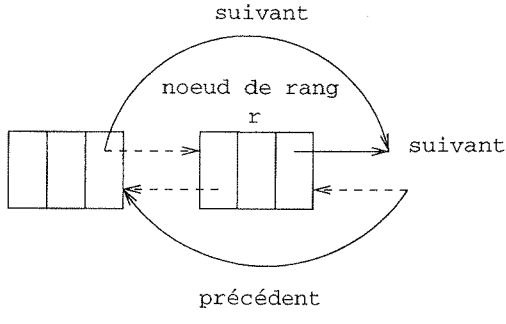


Figure 17.7 – *Suppression du nœud de rang r.*

L'ajout d'un élément est semblable à celui dans une liste simplement chaînée. Mais là aussi, il faut mettre à jour le lien `précédent` comme le montre la figure 17.8.

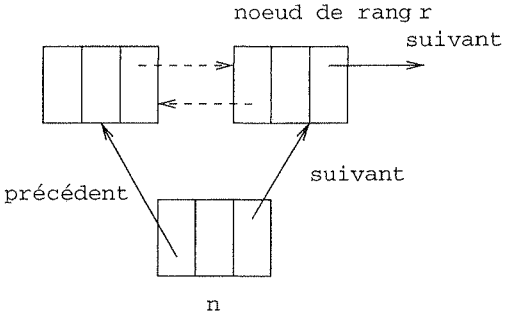


Figure 17.8 – *Ajout du nœud de rang r.*

La classe suivante donne l'écriture complète de l'implantation d'une liste avec une structure doublement chaînée.

```
public class ListeChaînéeDouble implements Liste {
   protected int lg;
   protected Noeud2 tête, queue;
   protected Class typeDesÉléments;
   public ListeChaînéeDouble(Class c) { typeDesÉléments=c; }
   public int longueur() { return lg; }
   public Object ième(int r) throws RangInvalideException {
       if (r<1 || r> lg) throw new RangInvalideException();
       Noeud2 n=tête;
       for (int i=1; i< r; i++) n=n.noeud2Suivant();
       // n désigne le noeud2 de rang r
       return n.valeur();
   }
   public void supprimer(int r) throws RangInvalideException {
       if (r<1 || r>lg) throw new RangInvalideException();
       if (lg==1) // un seul élément ⇒ r=1
           tête=queue=null;
       else // au moins 2 éléments
           if (r==1) // suppression en tête de liste
               tête=tête.noeud2Suivant();
           else
               if (r==lg)  {
                   // suppression du dernier élément de la liste
                   queue=queue.noeud2Précédent();
                   queue.suivant(null);
               }
               else { // cas général, r > 1 et r < lg
                   Noeud2 q=tête, p=null;
                   for (int i=1; i<r; i++) {
                       p=q;
                       q=q.noeud2Suivant();
```

```
                }
                // q désigne l'élément de rang r
                q.noeud2Suivant().précédent(p);
                p.suivant(q.suivant());
            }
        lg--;
    }
    public void ajouter(int r, Object e) throws RangInvalideException {
        if (e.getClass()!=typeDesÉléments)
            throw new TypeIncompatibleException();
        if (r<1 || r>lg +1)
            throw new RangInvalideException();
        Noeud2 n=new Noeud2(e);
        if (lg==0) // liste vide ⇒ r==1
            tête=queue=n;
        else
            if (r==1) { // insertion en tête de liste
                tête.précédent=n;
                n.suivant(tête);
                tête=n;
            }
            else
                if (r==lg+1)  {
                    // ajout du dernier élément de la liste
                    queue.suivant(n);
                    n.précédent(queue);
                    queue=n;
                }
                else { // cas général, r>1 et r ⩽ lg
                    Noeud2 p=null, q=tête;
                    for (int i=1; i<r; i++) {
                        p=q;
                        q=q.noeud2Suivant();
                    }
                    // q désigne l'élément de rang r
                    // get p son prédécesseur
                    p.suivant(n);
                    n.précédent(p);
                    n.suivant(q);
                    q.précédent(n);
                }
        lg++;
    }
} // fin classe ListeChaînéeDouble
```

17.1.3 Énumération

Il arrive fréquemment qu'on ait à parcourir une liste pour appliquer un traitement spécifique à chacun de ses éléments ; par exemple, pour afficher tous les éléments de la liste, ou encore

élever au carré chaque entier d'une liste. L'algorithme de parcours de la liste s'écrit de la façon suivante :

```
Algorithme parcours(l, traiter)
   pourtout i de 1 à longueur(l) faire
      traiter(ième(l,i))
   finpour
```

Il est évident que la complexité de cet algorithme doit être $\mathcal{O}(n)$. C'est le cas, si la liste est implantée avec un tableau. Mais, elle est malheureusement égale à $\mathcal{O}(n^2)$ si la liste utilise une structure chaînée, puisque l'opération `ième` repart à chaque fois du début de la liste.

Une des propriétés fondamentales des structures linéaires est que chaque élément, hormis le dernier, possède un successeur. Il est alors possible d'*énumérer* tous les éléments d'une liste grâce à une fonction *succ* dont la signature est définie comme suit :

$$\text{succ} : \mathcal{E} \rightarrow \mathcal{E}$$

Pour une liste l, cette fonction est définie par l'axiome suivant :

$$\forall r \in [1, \text{longueur}(l)[,\ \text{succ}(\text{ième}(l,r)) = \text{ième}(l,r+1)$$

L'implantation en JAVA

Nous représenterons une énumération par un objet qui implante l'interface suivante :

```
public interface Énumération {
  public Object élémentSuivant() throws FinÉnumérationException;
  public boolean finÉnumération();
}
```

La méthode `élémentSuivant` retourne l'élément suivant de l'énumération. La méthode `finÉnumération` indique si la fin de l'énumération a été atteinte ou pas. Notez que la liste n'est pas le seul type abstrait dont on peut énumérer les composants. Tout type abstrait qui possède cette propriété devra proposer une méthode qui retourne une énumération selon ses particularités. Une liste définira par exemple la méthode `listeÉnumération` suivante :

```
public Énumération listeÉnumération() {
        return new ListeÉnumération();
}
```

La programmation de la classe `ListeÉnumération` dépend de l'implantation de la liste. Chaque classe qui implante le type abstrait *Liste* définira une classe privée locale `ListeÉnumération` qui donne une implantation particulière de l'énumération. La classe `ListeTableau` définira :

```
private class ListeÉnumération implements Énumération {
  private int courant;
  private ListeÉnumération() {
       courant=0;
  }
```

```
  public boolean finÉnumération() {
       return courant==lg;
  }
  public Object élémentSuivant() throws FinÉnumérationException
  {
       if (courant==lg) throw new FinÉnumérationException();
       return éléments[courant++];
  }
} // ListeÉnumération
```

L'attribut `courant` désigne l'indice du prochain élément de l'énumération dans le tableau. Sa valeur initiale est égale à zéro.

Pour la classe `ListeTableauCirculaire`, le calcul de l'élément suivant se fait modulo la longueur du tableau.

```
private class ListeÉnumération implements Énumération {
   private int courant, nbÉnum;
   private ListeÉnumération() {
        courant=tête;
        nbÉnum=0;
   }
   public boolean finÉnumération() {
        return nbÉnum==lg;
   }
   public Object élémentSuivant() throws FinÉnumérationException
   {
        if (nbÉnum==lg) throw new FinÉnumérationException();
        Object e=éléments[courant];
        courant = courant==éléments.length-1 ? 0 : ++courant;
        nbÉnum++;
        return e;
   }
} // ListeÉnumération
```

L'attribut `nbÉnum` est nécessaire pour identifier la fin de la liste, car si les indices de tête et de queue sont égaux, la liste peut être aussi bien vide que pleine.

Pour les classes qui implantent les listes à l'aide de structures chaînées, il suffit de conserver une référence sur l'élément courant. L'accès au successeur se fait à l'aide de la méthode `noeudSuivant`. La première fois, sa valeur est égale à la tête de liste.

```
private class ListeÉnumération implements Énumération {
    private Noeud courant;
    private ListeÉnumération() {
        courant=tête;
    }
    public boolean finÉnumération() {
       return courant==null;
    }
    public Object élémentSuivant()
       throws FinÉnumérationException
```

```
    {
        if (courant==null)
                throw new FinÉnumérationException();
        Object e=courant.valeur();
        courant=courant.noeudSuivant();
        return e;
    }
} // ListeÉnumération
```

L'algorithme de parcours donné plus haut s'écrit en JAVA de la façon suivante. On crée d'abord l'énumération, puis on traite les éléments *un à un* grâce à la fonction `élémentSuivant`.

```
public void parcours(Liste l) {
        Énumération énum=l.listeÉnumération();
        while (! énum.finÉnumération())
              traiter(énum.élémentSuivant());
}
```

La manipulation d'une énumération s'applique en général à tous ses éléments. Il peut être en particulier très risqué de modifier la structure de données (*e.g.* la liste) au cours d'un parcours dans la mesure où cela peut corrompre l'énumération.

17.2 LES PILES

Une pile est une séquence d'éléments accessibles par une seule extrémité appelée *sommet*. Toutes les opérations définies sur les piles s'appliquent à cette extrémité. L'élément situé au sommet s'appelle le *sommet de pile*. La séquence formée de quatre entiers $< 5 \; -13 \; 23 \; 100 >$ est représentée sous forme de pile par la figure 17.9.

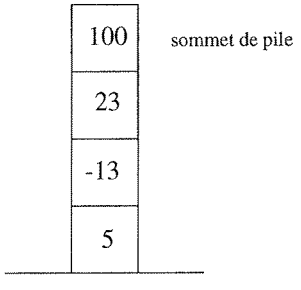


Figure 17.9 – *Une pile de quatre entiers.*

L'ajout et la suppression d'éléments en sommet de pile suivent le modèle *dernier entré – premier sorti* (LIFO [1]). Les piles sont des structures fondamentales, et leur emploi dans les programmes informatiques est très fréquent. Nous avons déjà vu que le mécanisme d'appel des sous-programmes suit ce modèle de pile. Les logiciels qui proposent une fonction « undo » se servent également d'une pile pour défaire, en ordre inverse, les dernières actions

1. *Last-In First-Out.*

effectuées par l'utilisateur. Les piles sont également nécessaires pour évaluer des expressions postfixées.

17.2.1 Définition abstraite

Ensembles

Pile est l'ensemble des piles dont les éléments appartiennent à un ensemble $\mathcal{E}$ quelconque. Les opérations sur les piles seront les mêmes quelle que soit la nature des éléments manipulés. La constante *pilevide* représente une pile vide.

Pile **utilise** $\mathcal{E}$ et booléen
pilevide $\in$ *Pile*

Description fonctionnelle

Quatre opérations abstraites sont définies sur le type *Pile* :

empiler	:	*Pile* $\times$ $\mathcal{E}$	$\rightarrow$	*Pile*
dépiler	:	*Pile*	$\rightarrow$	*Pile*
sommet	:	*Pile*	$\rightarrow$	$\mathcal{E}$
est-vide?	:	*Pile*	$\rightarrow$	booléen

Le rôle de l'opération *empiler* est d'ajouter un élément en sommet de pile, celui de *dépiler* de supprimer le sommet de pile et celui de *sommet* de retourner l'élément en sommet de pile. Enfin, l'opération *est-vide?* indique si une pile est vide ou pas.

Description axiomatique

La sémantique des fonctions précédentes est définie formellement par les axiomes suivants :

$\forall p \in$ *Pile*,$\forall e \in \mathcal{E}$
(1) est-vide?(pilevide) = vrai
(2) est-vide?(empiler(p,e)) = faux
(3) dépiler(empiler(p,e)) = p
(4) sommet(empiler(p,e)) = e
(5) $\nexists p$, p = dépiler(pilevide)
(6) $\nexists e$, e = sommet(pilevide)

Notez que ce sont les axiomes (3) et (4) qui définissent le comportement LIFO de la pile. Les opérations *dépiler* et *sommet* sont des fonctions partielles, et les axiomes (5) et (6) précisent leur domaine de définition ; ces deux opérations ne sont pas définies sur une pile vide.

17.2.2 L'implantation en Java

La définition fonctionnelle du type abstrait *Pile* est traduite par l'interface suivante :

```
public interface Pile {
    public boolean estVide();
    public Object sommet() throws PileVideException;
    public void dépiler() throws PileVideException;
    public void empiler(Object e);
}
```

Si elles opèrent sur une pile vide, les méthodes `dépiler` et `sommet` émettent l'exception `PileVideException`. Cette exception est simplement définie par la déclaration :

```
public class PileVideException extends RuntimeException {
    public PileVideException() {
        super();
    }
}
```

Les piles sont des listes particulières qui se distinguent par les méthodes d'accès aux éléments. Il est alors naturel de réutiliser, par héritage, les classes qui implémentent l'interface `Liste`. Par la suite, et pour des raisons de lisibilité, nous considérerons que les classes d'implantation des listes définissent les méthodes suivantes :

```
Object élémentDeTête() { return ième(1); }
Object élémentDeQueue() { return ième(lg); }
void ajouterEnTête(Object e) { ajouter(1,e); }
void ajouterEnQueue(Object e) { ajouter(lg+1,e); }
void supprimerEnTête() { supprimer(1); }
void supprimerEnQueue() { supprimer(lg); }
```

La complexité des opérations de pile est $\mathcal{O}(1)$ quelle que soit l'implantation choisie, tableau ou structure chaînée. L'implantation doit donc assurer un accès direct au sommet de la pile.

Utilisation d'un tableau

La figure 17.10 montre la séquence $< 5\ -13\ 23\ 100 >$ mémorisée dans un tableau. Pour repérer à tout moment le sommet de pile, il suffit d'un seul indice de queue.

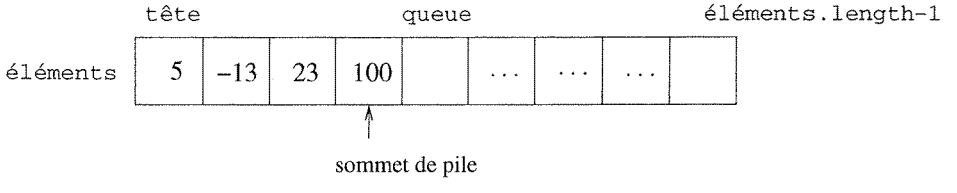


Figure 17.10 – *Une pile implantée par un tableau.*

Les algorithmes des opérations de pile sont très simples. L'opération *sommet* consiste à retourner l'élément de queue, alors que *dépiler* et *empiler* consistent, respectivement, à supprimer et à ajouter en queue.

Pour implanter la classe `PileTableau`, une gestion simple (*i.e.* non circulaire) d'une liste en tableau suffit. Cette classe est déclarée comme suit :

```
public class PileTableau extends ListeTableau implements Pile
{
    public PileTableau(Class c) { this(MAXÉLÉM, c); }
    public PileTableau(int n, Class c) {
        super(n,c);
    }
    public boolean estVide() {
      return longueur()==0;
    }
    public Object sommet() throws PileVideException {
        if (estVide()) throw new PileVideException();
        return élémentDeQueue();
    }
    public void empiler(Object e) {
        if (estPleine()) throw new PilePleineException();
        insérerEnQueue(e);
    }

    public void dépiler() throws PileVideException {
        if (estVide())
            throw new PileVideException();
        supprimerEnQueue();
    }
}
```

Notez l'utilisation de la fonction `estPleine` propre à l'implantation avec tableau. Cette fonction est héritée de la classe `listeTableau` et n'appartient pas à l'interface `Liste`.

```
protected boolean estPleine() {
    return lg==éléments.length;
}
```

Utilisation d'une structure chaînée

L'implantation d'une pile à l'aide d'une structure chaînée utilise la classe `ListeChaînée` qui ne nécessite qu'une référence sur la tête de liste. La figure 17.11 montre la séquence $< 5 \; -13 \; 23 \; 100 >$.

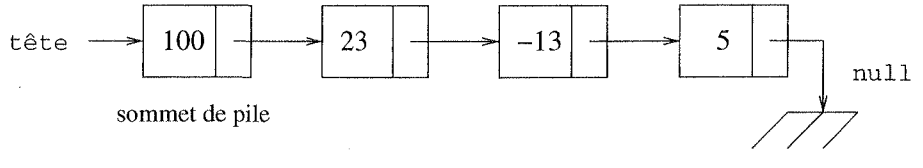


Figure 17.11 – *Une pile implantée par une structure chaînée.*

Pour garder leur complexité $\mathcal{O}(1)$, les opérations devront travailler sur la tête de liste à l'aide des méthodes `élémentDeTête`, `insérerEnTête` et `supprimerEnTête`.

```
public class PileChaînée extends ListeChaînée implements Pile
{
```

```
    public PileChaînée(class c) {
        super(c);
    }
    public boolean estVide() {
      return longueur()==0;
    }
    public Object sommet() throws PileVideException {
        if (estVide()) throw new PileVideException();
        return élémentDeTête();
    }
    public void empiler(Object e) {
        insérerEnTête(e);
    }
    public void dépiler() throws PileVideException {
        if (estVide())
            throw new PileVideException();
        supprimerEnTête();
    }
} // fin classe PileChaînée
```

17.3 LES FILES

Les files définissent le modèle *premier entré – premier sorti* (FIFO[2]). Les éléments sont insérés dans la séquence par une des extrémités et en sont extraits par l'autre. Ce modèle correspond à la file d'attente que l'on rencontre bien souvent face à un guichet dans les bureaux de poste, ou à une caisse de supermarché la veille d'un week-end. À tout moment, seul le premier client de la file accède au guichet ou à la caisse.

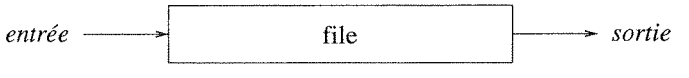


Figure 17.12 – *Une file.*

Le modèle de file est très utilisé en informatique. On le retrouve dans de nombreuses situations, comme, par exemple, dans la file d'attente d'un gestionnaire d'impression d'un système d'exploitation.

17.3.1 Définition abstraite

Ensembles

$\mathcal{F}ile$ est l'ensemble des files dont les éléments appartient à l'ensemble $\mathcal{E}$, et la constante *filevide* représente une file vide.

$\mathcal{F}ile$ **utilise** $\mathcal{E}$ et booléen
filevide $\in \mathcal{F}ile$

2. *First-In First-Out.*

Description fonctionnelle

Quatre opérations sont définies sur le type $\mathcal{F}ile$:

enfiler	:	$\mathcal{F}ile \times \mathcal{E}$	$\rightarrow$	$\mathcal{F}ile$
défiler	:	$\mathcal{F}ile$	$\rightarrow$	$\mathcal{F}ile$
premier	:	$\mathcal{F}ile$	$\rightarrow$	$\mathcal{E}$
est-vide?	:	$\mathcal{F}ile$	$\rightarrow$	booléen

L'opération *enfiler* a pour rôle d'ajouter un élément en queue de file, et l'opération *défiler* supprime l'élément en tête de file. *Premier* retourne le premier élément de la file et *est-vide?* indique si une file est vide ou pas. Notez que les signatures de ces opérations sont, au mot « file » près, identiques à celles des opérations du type abstrait $\mathcal{P}ile$. Ce sont bien les axiomes qui vont différencier ces deux types abstraits.

Description axiomatique

Ce sont en particulier, les axiomes (3) et (4), d'une part, et (5) et (6) d'autre part, qui distinguent le comportement de la file de celui de la pile. Ils indiquent clairement qu'un élément est ajouté par une extrémité de la file, et qu'il est accessible par l'autre extrémité.

$\forall f \in \mathcal{F}ile, \forall e \in \mathcal{E}$

(1) est-vide?(filevide) = vrai

(2) est-vide?(enfiler(f,e)) = faux

(3) est-vide?(f) $\Rightarrow$ premier(enfiler(f,e)) = e

(4) non est-vide?(f) $\Rightarrow$ premier(enfiler(f,e)) = premier(f)

(5) est-vide?(f) $\Rightarrow$ défiler(enfiler(f,e)) = filevide

(6) non est-vide?(f) $\Rightarrow$ défiler(enfiler(f,e)) = enfiler(défiler(f),e)

(7) $\nexists f$, f = défiler(filevide)

(8) $\nexists e$, e = premier(pilevide)

17.3.2 L'implantation en Java

L'interface suivante décrit les signatures des opérations du type $\mathcal{F}ile$.

```
public interface File {
    public boolean estVide();
    public Object premier() throws FileVideException;
    public void défiler() throws FileVideException;
    public void enfiler(Object e);
}
```

Lorsqu'elles opèrent sur des files vides, les méthodes `premier` et `défiler` émettent l'exception `FileVideException`.

Comme pour celle de `Pile`, l'implantation de l'interface `File` s'appuie sur les opérations proposées par le type `Liste`. Quelle que soit l'organisation des données choisie,

tableau ou structure chaînée, les algorithmes des opérations de `File` sont les mêmes. L'opération *premier* retourne l'élément de tête, *défiler* le supprime, alors que l'opération *enfiler* ajoute un élément en queue. Pour que ces opérations gardent une complexité égale à $\mathcal{O}(1)$, les classes `FileTableau` et `FileChaînée` devront utiliser, respectivement, les classes `ListeTableauCirculaire` et `ListeChaînéeDouble`.

```
public class FileTableau extends ListeTableauCirculaire
implements File
{
    public FileTableau(Class c) { super(c); }
    public FileTableau(int n, Class c) { super(n,c); }
    public boolean estVide() {
        return longueur()==0;
    }
    public Object premier() throws FileVideException {
        if (estVide())
            throw new FileVideException();
        return élémentDeTête();
    }
    public void enfiler(Object e) {
        if (estPleine())
            throw new FilePleineException();
        insérerEnQueue(e);
     }
    public void défiler() throws FileVideException {
        if (estVide())
            throw new FileVideException();
        supprimerEnTête();
    }
}
```

17.4 LES DÈQUES

Une dèque[3] possède à la fois les propriétés d'une pile et d'une file. On peut donc ajouter et supprimer un élément à chaque extrémité de la séquence. Les éléments de la séquence sont accessibles par les deux extrémités.

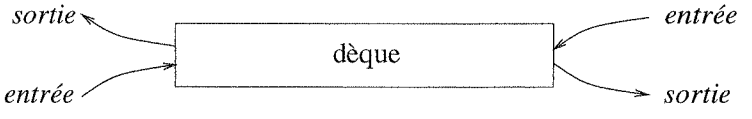


Figure 17.13 – *Une dèque.*

3. Le mot *dèque* vient de l'anglais « double ended queue ».

17.4.1 Définition abstraite

Ensembles

$\mathcal{D}ème que$ est l'ensemble des dèques dont les éléments appartiennent à un ensemble $\mathcal{E}$ quelconque. La constante *dèquevide* représente une dèque vide. L'ensemble $\mathcal{S}ens$ = {gauche, droite} est défini pour désigner l'extrémité utilisée par les différentes opérations de manipulation de dèque.

$\mathcal{D}ème que$ **utilise** $\mathcal{E}$, $\mathcal{S}ens$ et booléen
dèquevide $\in \mathcal{D}ème que$

Description fonctionnelle

Quatre opérations sont définies sur le type $\mathcal{D}ème que$:

endéquer	:	$\mathcal{D}ème que \times \mathcal{E} \times \mathcal{S}ens$	$\rightarrow$	$\mathcal{D}ème que$
dédéquer	:	$\mathcal{D}ème que \times \mathcal{S}ens$	$\rightarrow$	$\mathcal{D}ème que$
extrémité	:	$\mathcal{D}ème que \times \mathcal{S}ens$	$\rightarrow$	$\mathcal{E}$
est-vide?	:	$\mathcal{D}ème que$	$\rightarrow$	booléen

Description axiomatique

Les axiomes qui décrivent le type $\mathcal{D}ème que$ sont l'union des axiomes des types $\mathcal{P}ile$ et $\mathcal{F}ile$:

$\forall d \in \mathcal{D}ème que, \forall s, s1, s2 \in \mathcal{S}ens$ et $\forall e \in \mathcal{E}$

(1) est-vide?(dèquevide) = vrai
(2) est-vide?(endéquer(d,e,s)) = faux
(3) extrémité(endéquer(d,e,s),s) = e
(4) est-vide?(d) $\Rightarrow$ extrémité(endéquer(d,e,$s1$),$s2$) = e
(5) non est-vide?(d) $\Rightarrow$ extrémité(endéquer(d,e,$s1$),$s2$) = extrémité(d,$s2$)
(6) dédéquer(endéquer(d,e,s),s) = e
(7) est-vide?(d) $\Rightarrow$ dédéquer(endéquer(d,e,s),s) = d
(8) non est-vide?(d) $\Rightarrow$
dédéquer(endéquer(d,e,$s1$),$s2$) = endéquer(dédéquer(d,$s2$),t,$s1$)
(9) $\nexists d$, d = dédéquer(dèquevide ,s)
(10) $\nexists e$, e = extrémité(dèquevide ,s)

17.4.2 L'implantation en Java

L'interface `Dèque` suivante décrit les signatures des opérations du type abstrait $\mathcal{D}ème que$. Dans la mesure où le langage JAVA n'offre pas à l'utilisateur la possibilité de définir ses propres types élémentaires, nous représenterons le type $\mathcal{S}ens$ par les deux constantes entières `GAUCHE` et `DROITE`.

```
public interface Dèque {
    public static final int GAUCHE=0;
    public static final int DROITE=1;
    public boolean estVide();
    public Object extrémité(int sens) throws DèqueVideException;
    public void dédéquer(int sens) throws DèqueVideException;
    public void endéquer(Object e, int sens);
}
```

Leurs algorithmes de mise en œuvre du type `Dèque` sont très simples et leur programmation utilise les opérations du type `Liste`. Pour que la complexité des opérations soit $\mathcal{O}(1)$, l'implantation de l'interface `Dèque` devra choisir un tableau géré de façon circulaire ou une structure doublement chaînée.

```
public class DèqueChaînée extends ListeChaînéeDouble
implements Dèque
{
    public DèqueChaînée(Class c) { super(c); }
    public boolean estVide() {
        return longueur()==0;
    }
    public void dédéquer(int sens) throws DèqueVideException
    {
        if (estVide()) throw new DèqueVideException();
        if (sens==GAUCHE) supprimerEnQueue();
        else // sens=DROITE
            supprimerEnTête();
    }
    public void endéquer(Object e, int sens) {
        if (sens==GAUCHE) insérerEnQueue(e);
        else // sens=DROITE
            insérerEnTête(e);
    }
    public Object extrémité(int sens) throws DèqueVideException {
        if (estVide()) throw new DèqueVideException();
        return sens==GAUCHE ? élémentDeQueue()
                            : élémentDeTête();
    }
}
```

17.5 EXERCICES

Exercice 17.1. On veut enrichir le type abstrait *Liste* avec les opérations *concaténer* et *inverser*. Leurs signatures sont les suivantes :

concaténer : $Liste \times Liste \rightarrow Liste$
inverser : $Liste \rightarrow Liste$

Donnez les axiomes qui définissent la sémantique de ces opérations. Écrivez les méthodes `concaténer` et `inverser` qui respectent les définitions fonctionnelles et axiomatiques précédentes.

Exercice 17.2. Vous avez remarqué que l'implantation des opérations *ajouter* et *enlever* avec une structure simplement chaînée doit tenir compte du cas particulier de la modification de la référence sur l'élément de tête. Par exemple, à chaque ajout l'opération vérifie systématiquement si l'élément est à insérer en tête de liste ou pas. Il est possible d'éviter ce test si on considère qu'une liste vide contient toujours un élément. Cet élément, sans valeur particulière, est appelé *élément bidon*. Récrivez les opérations de la classe `ListeChaînée` en gérant un élément bidon.

Exercice 17.3. Le problème évoqué dans l'exercice précédent se pose également avec l'élément de fin d'une liste doublement chaînée. Récrivez les opérations de la classe `ListeChaînéeDouble` en gérant deux éléments bidons, respectivement, en tête et en queue de liste.

Exercice 17.4. Utilisez une liste pour représenter un polynôme à une variable de degré n. Vous programmerez les opérations telles que l'addition et la multiplication de deux polynômes.

Exercice 17.5. On désire évaluer des expressions postfixées formées d'opérandes entiers positifs et des quatre opérateurs `+`, `-`, `*` et `/`. On rappelle que dans la notation polonaise inversée l'opérateur suit ses opérandes. Par exemple, l'expression infixe suivante :

```
(7 + 2) * (5 - 3)
```

est dénotée :

```
7 2 + 5 3 - *
```

L'évaluation d'une expression postfixée se fait très simplement à l'aide d'une pile. L'expression est lue de gauche à droite. Chaque opérande lu est empilé et chaque opérateur trouve ses deux opérandes en sommet de pile qu'il remplace par le résultat de son opération. Lorsque l'expression est entièrement lue, sans erreur, la pile ne contient qu'une seule valeur, le résultat de l'évaluation. Écrivez l'algorithme d'évaluation d'une expression postfixée.

Programmez en JAVA cet algorithme, en utilisant une classe d'implantation du type *Pile*. Les expressions sont lues sur l'entrée standard `System.in` et les résultats sont écrits sur la sortie standard `System.out`. Vous traiterez un opérateur supplémentaire, dénoté par le symbole `=`, qui affiche le résultat. Les opérateurs et les opérandes sont séparés par des blancs, des tabulations ou des passages à la ligne. Vous pourrez utiliser la classe `StreamTokenizer` définie dans le paquetage `java.io`. Un objet de cette classe prendra en entrée un flot de caractères et rendra en sortie un flot d'*unités syntaxiques* qui représente les différents composants (opérandes, opérateurs, séparateurs) d'une expression.

Exercice 17.6. La définition du type abstrait *Liste*, donnée dans ce chapitre, suit un modèle itératif. Mais, il est également possible d'en donner une définition récursive :

$$\mathcal{L}iste = \emptyset + \mathcal{E} \times \mathcal{L}iste$$

Elle énonce qu'une liste est soit vide, soit formée d'un élément suivi d'une liste. On définit les opérations suivantes :

tête :	$\mathcal{L}iste$	$\rightarrow \mathcal{E}$
fin :	$\mathcal{L}iste$	$\rightarrow \mathcal{L}iste$
cons :	$\mathcal{E} \times \mathcal{L}iste$	$\rightarrow \mathcal{L}iste$
est-vide? :	$\mathcal{L}iste$	$\rightarrow$ booléen

L'opération *tête* retourne le premier élément de la liste, et *fin* la liste amputée du premier élément. L'opération *cons* construit une liste à partir d'un élément (à placer en tête) et d'une liste.

Les algorithmes de manipulation de cette forme de liste sont naturellement récursifs. Par exemple, parcourir une liste pour appliquer un traitement sur chacun des éléments s'écrit :

```
Algorithme appliquer(l, traiter)
   si non est-vide?(l) alors
       traiter(tête(l))
       appliquer(fin(l))
   finsi
```

Écrivez les axiomes qui définissent la sémantique des opérations précédentes. Proposez une implantation en JAVA de ce modèle de $\mathcal{L}iste$ à l'aide d'une structure chaînée.

Exercice 17.7. Une matrice creuse est une matrice dont la majorité des éléments sont égaux à zéro. Proposez une représentation d'une matrice creuse à l'aide d'une structure chaînée qui ne mémorise que les valeurs différentes de zéro.

Chapitre **18**

Graphes

Un graphe est un ensemble de *sommets* reliés par des . La figure 18.1 montre un graphe particulier à neuf sommets et dix arcs. Par définition, un graphe est *orienté*, c'est-à-dire que les relations établies entre les sommets ne sont pas symétriques. Toutefois, certains problèmes, qui ne tiennent pas compte de l'orientation, pourront les considérer comme symétriques. On parle alors de graphe *non orienté* et les arcs qui relient les sommets sont nommés *arêtes*.

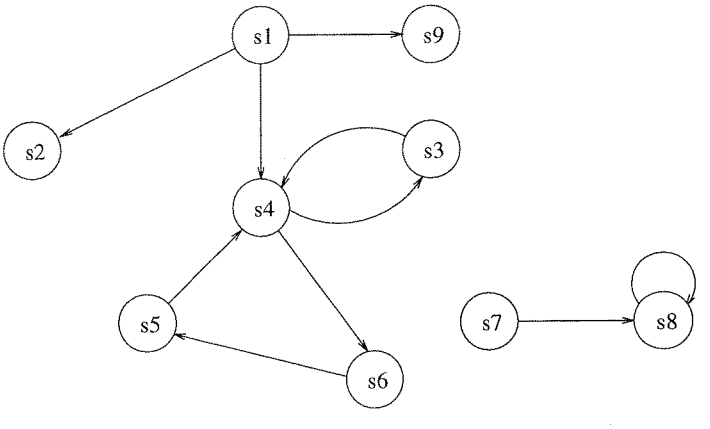


Figure 18.1 – *Un graphe à neuf sommets et deux composantes connexes.*

Les graphes interviennent dans des domaines variés tant théoriques (*e.g.* mathématiques discrètes, combinatoire) que pratiques (*e.g.* applications informatiques), et sont très utilisés dès qu'il s'agit de simuler des relations complexes entre des éléments d'un ensemble.

Les graphes servent par exemple en sociologie à modéliser les relations entre des individus ou en sciences économiques. Naturellement, ce sont les outils de prédilection pour la représentation des réseaux (routiers, de télécommunication, d'interconnexion de réseaux, de processeurs, etc.). Par exemple, le réseau routier entre les grandes villes d'un pays peut être assimilé à un graphe non orienté. Un sommet représentera une ville et une arête une route entre deux villes.

En informatique, les objets alloués dynamiquement dans la zone de tas de la mémoire d'un ordinateur s'organisent en graphe, et sont gérés automatiquement par les récupérateurs de mémoire[1] de certains langages de programmation, comme c'est le cas en JAVA. La phase d'optimisation globale des compilateurs construit un graphe de *flot de contrôle* du programme à partir duquel elle améliorera le code cible à produire selon des techniques classiques de substitution d'appels de procédures, de réduction de puissance, etc. [ASU89].

Il est possible d'ajouter des informations sur les sommets pour les identifier, ou sur les arcs pour les pondérer. Les noms des villes seront associés aux sommets du graphe qui modélise le réseau routier, et on placera sur chaque arête la distance qui sépare deux villes. Avec ces informations, il sera possible, par exemple, de calculer le trajet le plus court pour se rendre d'une ville à une autre. Un graphe dont les arcs (ou les arêtes) portent des valuations est appelé graphe *valué*.

Il est impossible en quelques lignes introductives de présenter tous les domaines d'applications des graphes. Nous convions le lecteur intéressé à se reporter aux ouvrages suivants [Ber70, MB86, Gon95].

Dans ce chapitre, après avoir introduit quelques termes spécifiques aux graphes, nous présenterons un type abstrait *Graphe* et ses mises en œuvre possibles. Quelques algorithmes classiques sur les graphes seront donnés au chapitre 23.

18.1 TERMINOLOGIE

Un graphe $G = (X,U)$ est formé d'un ensemble de sommets X et d'un ensemble d'arcs U. L'*ordre* d'un graphe est son nombre de sommets. Les graphes *creux* ont peu d'arcs et ceux qui en possèdent beaucoup sont dits *denses*.

Un arc $u = (x,y) \in U$ possède un extrémité *initiale* x et une extrémité *finale* y. x est le prédécesseur de y et y est le successeur de x. Si $x = y$, l'arc est appelé une *boucle*. Une arête entre deux sommets est notée $e = [x,y]$. L'ensemble des *voisins* d'un arc est l'union de l'ensemble de ses prédécesseurs et de ses successeurs.

Un *multigraphe* est un graphe qui possède des boucles ou des arcs *multiples* (plusieurs arcs qui possèdent la même extrémité initiale et la même extrémité finale). Un graphe *simple* est un graphe sans boucle, ni arc multiple. Par la suite, nous ne considérerons que les graphes simples. Le nombre d'arcs d'un graphe simple à n sommets est compris entre 0 et $n(n-1)$, et $\frac{1}{2}n(n-1)$ si on ne considère pas l'orientation des arcs.

Deux arcs sont dits *adjacents* s'ils ont au moins une extrémité commune. Un arc u est *incident* à un sommet x *vers l'extérieur* si x est l'extrémité initiale de u. Le *demi-degré*

1. En anglais *garbage-collector*.

extérieur, noté $d^+(x)$, est le nombre d'arcs incidents vers l'extérieur à un sommet x. De même, un arc u est *incident* à un sommet y *vers l'intérieur* si y est l'extrémité finale de u. Le *demi-degré intérieur*, noté $d^-(x)$, est le nombre d'arcs incidents vers l'intérieur à un sommet x. Le *degré* d'un sommet x est égal à $d^-(x) + d^+(x)$.

Un graphe est *complet* s'il existe un arc (x,y) pour tout x et y de X. Un *sous-graphe* $G' = (A,U)$ d'un graphe $G = (X,U)$ est un graphe dont les arcs de U ont leurs extrémités dans A. Un graphe *partiel* $G' = (X,V)$ d'un graphe $G = (X,U)$ est un graphe dont les sommets de X sont les extrémités des arcs de V.

Un *chemin* est une liste de sommets dans lequel deux sommets successifs quelconques sont reliés par un arc. La *longueur* d'un chemin est égale au nombre d'arcs. Un chemin qui ne rencontre pas deux fois le même sommet est dit *élémentaire*. Un *cycle* est un chemin dont le premier et le dernier sommet sont identiques.

Un graphe est dit *connexe* s'il existe un chemin reliant toute paire de sommets. Il est *fortement* connexe si un tel chemin existe de x vers y *et* de y vers x. Un graphe non connexe peut être formé de *composantes (fortement) connexes*.

La *racine* d'un graphe est un sommet r tel que pour tout sommet y, il existe un chemin entre r et y.

18.2 DÉFINITION ABSTRAITE D'UN GRAPHE

➤ Ensembles

Graphe **utilise** *Sommet*, booléen
graphevide $\in$ *Graphe*

avec *Graphe*, l'ensemble des graphes orientés, et *Sommet* l'ensemble des sommets du graphe. La constante graphevide définit un graphe sans sommet.

➤ Description fonctionnelle

À partir des définitions données à la section 18.1, nous pouvons proposer les opérations suivantes :

ordre	:	*Graphe*	→	naturel
arc	:	*Graphe* × *Sommet* × *Sommet*	→	booléen
d^+	:	*Graphe* × *Sommet*	→	naturel
d^-	:	*Graphe* × *Sommet*	→	naturel
degré	:	*Graphe* × *Sommet*	→	naturel
ièmeSucc	:	*Graphe* × *Sommet* × naturel	→	*Sommet*
ajouterArc	:	*Graphe* × *Sommet* × *Sommet*	→	*Graphe*
supprimerArc	:	*Graphe* × *Sommet* × *Sommet*	→	*Graphe*
ajouterSommet	:	*Graphe* × *Sommet*	→	*Graphe*
enleverSommet	:	*Graphe* × *Sommet*	→	*Graphe*

L'opération *arc* teste s'il existe un arc entre deux sommets. Les opérations d^+ et d^- définissent, respectivement, le demi-degré extérieur et le demi-degré intérieur. La fonction *ièmeSucc* retourne le ième successeur d'un sommet.

Lorsqu'on considère un graphe non orienté, on ajoute au type abstrait les opérations suivantes :

arête	:	*Sommet* × *Sommet*	→	booléen
ajouterArête	:	*Graphe* × *Sommet* × *Sommet*	→	*Graphe*
supprimerArête	:	*Graphe* × *Sommet* × *Sommet*	→	*Graphe*

Enfin, pour les graphes valués, les opérations *ajouterArc* et *ajouterArête* possèdent les signatures suivantes :

ajouterArc	:	*Graphe* × *Sommet* × *Sommet* × $\mathcal{E}$	→	*Graphe*
ajouterArête	:	*Graphe* × *Sommet* × *Sommet* × $\mathcal{E}$	→	*Graphe*

avec $\mathcal{E}$ désignant un ensemble de valeurs quelconques. La valeur d'un arc ou d'une arête est obtenue grâce aux opérations suivantes :

valeurArc	:	*Graphe* × *Sommet* × *Sommet*	→	$\mathcal{E}$
valeurArête	:	*Graphe* × *Sommet* × *Sommet*	→	$\mathcal{E}$

Description axiomatique

$\forall g \in \mathit{Graphe}, \forall x,y \in \mathit{Sommet}$

(1) $x \in g \Rightarrow \nexists\, g',\ g' = \text{ajouterSommet}(g,x))$

(2) $\text{ordre}(\text{graphevide}) = 0$

(3) $\text{ordre}(\text{ajouterSommet}(g,x)) = \text{ordre}(g) + 1$ et $\text{degré}(x) = \text{d}^+(x) = \text{d}^-(x) = 0$

(4) $x \notin g \Rightarrow \nexists\, g',\ g = \text{enleverSommet}(g,x))$

(5) $\text{ordre}(\text{enleverSommet}(g,x)) = \text{ordre}(g) - 1$ et $\text{arc}(y,x) \Rightarrow \text{d}^+(y) = \text{d}^+(y) - 1$

(6) $\text{degré}(x) = \text{d}^+(x) + \text{d}^-(x)$

(7) $\text{arc}(x,y) \Rightarrow \nexists\, g',\ g' = \text{ajouterArc}(g,x,y))$

(8) $\text{ajouterArc}(g,x,y) \Rightarrow \text{d}^+(x) = \text{d}^+(x) + 1$ et $\text{d}^-(y) = \text{d}^-(y) + 1$

(9) $\text{non arc}(x,y) \Rightarrow \nexists\, g',\ g' = \text{supprimerArc}(g,x,y))$

(10) $\text{supprimerArc}(g,x,y) \Rightarrow \text{d}^+(x) = \text{d}^+(x) - 1$ et $\text{d}^-(y) = \text{d}^-(y) - 1$

(11) $\forall i \in [1, \text{d}^+(g,x)],\ \text{arc}(g,x,\text{ièmeSucc}(g,x,i)) = \text{vrai}$

(12) $\forall i \in [1, \text{d}^+(g,x)],\ y \neq \text{ièmeSucc}(g,x,i) \Rightarrow \text{non arc}(g,x,y)$

Lorsque l'orientation des arcs ne joue aucun rôle, on considère les opérations sur les arêtes.

(13) $\text{ajouterArête}(g,x,y) \Rightarrow \text{ajouterArc}(x,y)$ et $\text{ajouterArc}(y,x)$

(14) $\text{arête}(g,x,y) \Rightarrow \text{arc}(x,y)$ et $\text{arc}(y,x)$

18.3 L'IMPLANTATION EN JAVA

L'interface `Graphe` suivante donne les signatures des opérations du type abstrait *Graphe*.

```
public interface Graphe {
 public int ordre();
 public boolean arête(Sommet s1, Sommet s2);
 public boolean arc(Sommet s1, Sommet s2);
 public int demiDegréInt(Sommet s);
 public int demiDegréExt(Sommet s);
 public int degré(Sommet s);
 public Sommet ièmeSucc(Sommet s, int i);
 public void ajouterSommet(Sommet s) throws SommetException;
 public void enleverSommet(Sommet s) throws SommetException;
 public void ajouterArc(Sommet s1, Sommet s2) throws ArcException;
 public void supprimerArc(Sommet s1, Sommet s2) throws ArcException;
 public void ajouterArête(Sommet s1, Sommet s2) throws ArêteException;
 public void supprimerArête(Sommet s1, Sommet s2) throws ArêteException;
 public Énumération grapheÉnumération();
 public Énumération sommetsAdjacents(Sommet s);
}
```

Notez que l'interface définit deux opérations supplémentaires, `grapheÉnumération` et `sommetsAdjacents`. Ces méthodes retournent l'énumération, respectivement, de tous les sommets du graphe, et de tous les successeurs d'un sommet passé en paramètre. Ces méthodes seront très utiles dans la programmation des algorithmes de manipulation de graphe. En particulier `sommetsAdjacents` permettra un calcul de tous les successeurs plus efficace que :

```
{parcourir tous les successeurs de s dans g}
pourtout i de 1 à d+(g,s) faire
        succ ← ièmeSucc(g,s,i)
finpour
```

Un graphe est implanté classiquement soit par une *matrice d'adjacence*, soit par des *listes d'adjacence*. Le choix de la représentation d'un graphe sera guidé par sa densité, mais aussi par les opérations qui sont appliquées. D'une façon générale, plus le graphe est dense, plus la matrice d'adjacence conviendra. Au contraire, pour des graphes creux, les listes d'adjacences seront plus adaptées. Dans les sections suivantes, nous décrirons ces deux sortes de mises en œuvre, et les complexités des opérations seront exprimées pour un graphe d'ordre n.

18.3.1 Matrice d'adjacence

Une matrice d'adjacence $n \times n$, représentant un graphe à n sommets, possède des éléments booléens tels que $m[i,j] = vrai$ s'il existe un arc entre i et j et *faux* sinon. Avec cette représentation, la complexité en espace mémoire est $\mathcal{O}(n^2)$.

Le graphe de la figure 18.1 page 207 est représenté par la matrice d'adjacence suivante[2] :

	s1	s2	s3	s4	s5	s6	s7	s8	s9
s1	faux	vrai	faux	vrai	faux	faux	faux	faux	vrai
s2	faux	faux	faux	faux	faux	faux	faux	faux	faux
s3	faux	faux	faux	vrai	faux	faux	faux	faux	faux
s4	faux	faux	vrai	faux	faux	vrai	faux	faux	faux
s5	faux	faux	faux	vrai	faux	faux	faux	faux	faux
s6	faux	faux	faux	faux	vrai	faux	faux	faux	faux
s7	faux	faux	faux	faux	faux	faux	faux	vrai	faux
s8	faux	faux	faux	faux	faux	faux	faux	vrai	faux
s9	faux	faux	faux	faux	faux	faux	faux	faux	faux

Une classe `GrapheMatrice` qui implante l'interface `Graphe` peut utiliser les déclarations suivantes :

```
protected int nbSommets; // ordre du graphe
protected boolean [][] matI;
```

L'ensemble *Sommet* peut être quelconque, mais l'implantation doit nécessairement offrir une bijection entre le type des indices de la matrice et le type *Sommet*. Ainsi, les deux fonctions suivantes doivent être définies :

numéro : *Sommet* → int
sommet : int → *Sommet*

La fonction *numéro* retourne le numéro de l'indice d'un sommet dans la matrice d'adjacence, et la fonction *sommet* est sa réciproque.

Notez que pour un graphe non orienté la matrice est symétrique. Pour représenter un graphe valué, on choisit une matrice dont les éléments représentent la valeur de l'arc entre deux sommets.

L'utilisation de matrice d'adjacence est commode pour tester l'existence d'une arête ou d'un arc entre deux sommets. La complexité de ces opérations est $\mathcal{O}(1)$.

```
public boolean arc(Sommet s1, Sommet s2) {
    return matI[numéro(s1)][numéro(s2)];
}
```

En revanche, le calcul du ième successeur d'un sommet, ou celui de son demi-degré intérieur ou extérieur, nécessite n tests quel que soit le nombre de successeurs du sommet. La complexité est $\mathcal{O}(n)$.

```
public int demiDegréInt(Sommet s) {
    int nbDegrésInt=0;
```

2. Pour répérer facilement les arcs, les valeurs *vrai* sont encadrées.

```
    for (int i=0; i<nbSommets; i++)
        if (matI[i][numéro(s)]) nbDegrésInt++;
    return nbDegrésInt;
}

public int demiDegréExt(Sommet s) {
    int nbDegrésExt=0;
    for (int i=0; i<nbSommets; i++)
        if (matI[numéro(s)][i]) nbDegrésExt++;
    return nbDegrésExt;
}
public Sommet ièmeSucc(Sommet s, int i) {
    if (i<=0) throw new SommetException();
    int k=0;
    do {
       if (k==nbSommets) throw new SommetException();
       if (matI[s.numéro()][k++]) i--;
    } while (i!=0);
    // k est le numéro du ième successeur du sommet s
    return sommet(k-1);
}
```

La construction des énumérations de sommets suit le même type de programmation donnée à la section 17.1.3. Donnons, par exemple, la méthode d'énumération des successeurs d'un sommet.

```
public Énumération sommetsAdjacents(Sommet s) {
    return new SommetsAdjacentsÉnumération(s);
}
```

La classe `SommetsAdjacentsÉnumération` est une classe privée locale à `GrapheMatrice`. Son constructeur fabrique une liste de successeurs, et les méthodes `élémentSuivant` et `finÉnumération` permettent son énumération par réutilisation de l'énumération de liste.

```
private class SommetsAdjacentsÉnumération implements Énumération {
  private Énumération énumSommets;

  public SommetsAdjacentsÉnumération(Sommet s) {
     // construire la liste des successeurs de s
      Liste listeSom=new ListeChaînée2(Sommet.class);
      int i=0;
      do
         if (matI[numéro(s)][i])
             listeSom.ajouter(listeSom.longueur()+1,sommet(i));
      while (++i<nbSommets);
      énumSommets=listeSom.listeÉnumération();
  }
  public boolean finÉnumération() {
      return énumSommets.finÉnumération();
  }
```

```
  public Object élémentSuivant() throws FinÉnumérationException
  {
      if (énumSommets.finÉnumération())
          throw new FinÉnumérationException();
      return énumSommets.élémentSuivant();
  }
} // fin classe SommetsAdjacentsÉnumération
```

18.3.2 Listes d'adjacence

Cette représentation du graphe consiste à associer à chaque sommet la liste ordonnée de ses successeurs. Ces listes s'appellent des *listes d'adjacence*. La figure 18.2 montre le graphe de la page 207 représenté sous cette forme.

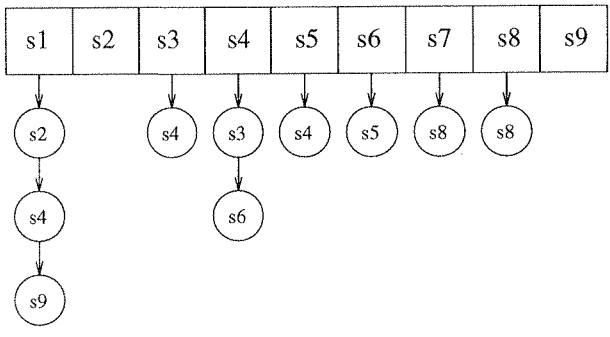


Figure 18.2 – *Le graphe de la figure 18.1 représenté par des listes d'adjacence.*

En réutilisant le type `Liste` défini au chapitre 17, il possible de représenter simplement le graphe par le tableau suivant :

```
protected Liste [] listeI;
```

Il est clair que cette représentation permet un gain de place substantiel pour des graphes creux, c'est-à-dire lorsque le nombre d'arcs est petit par rapport au nombre de sommets. Un graphe orienté de n sommets et p arcs nécessite $n + p$ éléments de liste, alors qu'un graphe non orienté en nécessite $n + 2p$.

Le calcul du demi-degré extérieur d'un sommet est immédiat puisqu'il est égal à la longueur de sa liste d'adjacence. Sa complexité est $\mathcal{O}(1)$.

```
public int demiDegréExt(Sommet s) {
    return listeI[numéro(s)].longueur();
}
```

En revanche, vérifier l'existence d'un arc entre deux sommets ou calculer un ième successeur nécessite le parcours d'une liste d'adjacence. La complexité est alors égale, en moyenne, à la longueur de la liste d'adjacence divisée par deux. La complexité dans le pire des cas est donc $\mathcal{O}(p)$.

```
public boolean arc(Sommet s1, Sommet s2) {
    return rechercher(listeI[numéro(s1)], s2);
}

public Sommet ièmeSucc(Sommet s, int i) {
    return (Sommet) listeI[numéro(s)].ième(i);
}
```

Le calcul du demi-degré intérieur d'un sommet nécessite quant à lui le parcours de l'ensemble des listes d'adjacence, c'est-à-dire le nombre total d'arcs du graphe. La complexité est $\mathcal{O}(\max(n,p))$.

Pour représenter un graphe valué, il suffit d'ajouter les valuations des arcs (x,y) à l'élément y dans la liste d'adjacence du sommet x. La figure 18.3 montre cette forme de représentation.

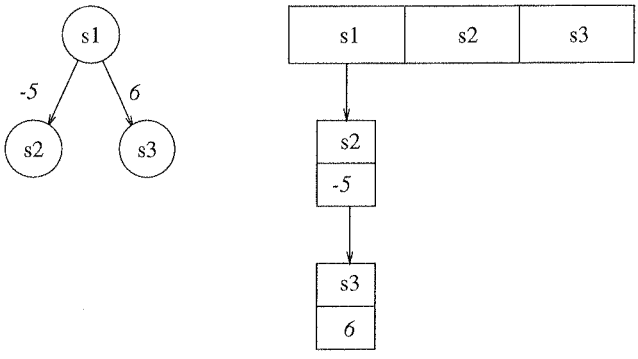


Figure 18.3 – *Représentation d'un graphe valué par des listes d'adjacence.*

L'énumération des successeurs d'un sommet s consiste simplement à énumérer les sommets de sa liste d'adjacence.

18.4 PARCOURS D'UN GRAPHE

Il existe deux types classiques de parcours de graphe. Le parcours en *profondeur* et le parcours en *largeur*. Ces deux types de parcours font un parcours complet du graphe, et visitent chacun des sommets, une seule fois, pour lui appliquer un traitement.

18.4.1 Parcours en profondeur

Le parcours en profondeur, à partir d'un sommet s, passe d'abord par ce sommet, puis consiste à parcourir en profondeur chacun de ses successeurs. Ce parcours correspond donc à celui d'une composante connexe du graphe. Si le graphe en possède plusieurs, tous les sommets n'auront pas été traités, et l'algorithme devra se poursuivre avec un parcours en profondeur d'un sommet non encore traité.

Le parcours peut passer plusieurs fois par un même sommet (à cause d'un cycle par exemple), et le traitement du sommet ne devra pas être renouvelé. Pour vérifier si un sommet a déjà été traité ou pas, on le marque après chaque traitement. L'algorithme s'exprime récursivement en deux parties comme suit :

```
Algorithme Parcours-en-Profondeur(G)
   {Parcours en profondeur du graphe G}
   pourtout s de G faire
        si non marqué(s) alors
            Pprofondeur(G, s)
        finsi
   finpour
└──────────────
```

```
Algorithme Pprofondeur(G, s)
   {Parcours en profondeur des successeurs du sommet s}
   mettre une marque sur s
   pourtout x de G tel que ∃ arc(s,x) faire
        si non marqué(x) alors
              Pprofondeur(G, x)
        finsi
   finpour
└──────────────
```

Si le traitement du sommet a lieu avant le parcours des successeurs le parcours est dit *préfixe*. Cela correspond à l'exécution d'une action sur le sommet courant juste avant la pose de la marque sur le sommet. En revanche, si le traitement est fait après le parcours des successeurs, le parcours est *postfixe*. L'action est faite sur le sommet courant après l'énoncé itératif. Notez que deux traitements, préfixe et postfixe, peuvent être appliqués lors d'un même parcours.

Les parcours en profondeur préfixe et postfixe du graphe de la figure 18.1 traitent les sommets selon les ordres suivants :

préfixe = <s1 s2 s4 s3 s6 s5 s9 s7 s8>
postfixe = <s2 s3 s5 s6 s4 s9 s1 s7 s8>

La complexité du parcours en profondeur dépend de la représentation du graphe. Pour un graphe de n sommets et p arcs, la complexité de l'algorithme est $\mathcal{O}(n^2)$ avec une matrice d'adjacence, et $\mathcal{O}(\max(n,p))$ avec des listes d'adjacence.

18.4.2 Parcours en largeur

On appelle « distance » la longueur du chemin entre deux sommets d'un graphe. Le parcours en largeur d'un graphe à partir d'un sommet origine s consiste d'abord à visiter ce sommet, puis à traiter les sommets de distance avec s égale à un, puis ceux de distance égale à deux, etc. L'algorithme s'écrit de façon itérative et utilise une file d'attente qui conserve les sommets déjà traités par ordre de distance croissante dont les successeurs sont à visiter. Comme pour le parcours en profondeur, les nœuds parcourus sont marqués afin de ne pas les traiter plusieurs fois. Notez qu'un parcours en largeur parcourt une composante connexe du

graphe. Pour un parcours complet d'un graphe à plusieurs composantes connexes, on décrit l'algorithme en deux étapes comme précédemment :

```
Algorithme Parcours-en-Largeur(G)
   {Parcours en largeur de toutes les composantes connexes du
    graphe G}
   pourtout s de G faire
        si non marqué(s) alors
            Plargeur(G, s)
        finsi
   finpour
```

```
Algorithme Plargeur(G, s)
   {Parcours en largeur des successeurs du sommet s}
   mettre une marque sur s
   f ← filevide
   enfiler(f,s)
   tantque non estvide(f) faire
        p ← premier(f)
        défiler(f)
        pourtout x de G tel que ∃ arc(p,x) faire
             si non marqué(x) alors
                mettre une marque sur x
                enfiler(f,x)
             insi
        finpour
   fintantque
```

À partir du sommet s1, le parcours en largeur du graphe de la figure 18.1 traite les sommets dans l'ordre suivant :

<s1 s2 s4 s9 s3 s6 s5 s7 s8>

La complexité du parcours en largeur est identique à celle d'un parcours en profondeur, quelle que soit la représentation choisie matrice d'adjacence ou listes d'adjacence.

18.4.3 Programmation en Java des parcours de graphe

Les algorithmes de parcours sont implantés par trois méthodes dont les signatures complètent les interfaces `Graphe` et `GrapheValué`.

```
public void parcoursProfondeurPréfixe(Opération op);
public void parcoursProfondeurPostfixe(Opération op);
public void parcoursLargeur(Opération op);
```

Le traitement effectué sur chacun des sommets est contenu dans le paramètre `op` de type `Opération`. C'est une interface qui déclare l'opération générique `exécuter` qui prend en

paramètre une donnée de type `Object` sur laquelle s'applique l'opération, et qui renvoie un résultat lui-aussi de type `Object`.

```
public interface Opération {
      public Object exécuter(Object e);
}
```

Lors d'un parcours effectif d'un graphe, on transmet à la méthode de parcours un objet qui donne une implantation particulière de l'interface `Opération`. Par exemple, pour afficher tous les sommets d'un graphe sur la sortie standard, on pourra définir la classe :

```
public class OpérationAfficher implements Opération {
    public Object exécuter(Object e) {
        System.out.print(e);
        return null;
    }
}
```

Nous donnons la programmation du parcours en largeur. Elle suit l'algorithme de la page 217, et ne pose guère de difficultés. On représente les marques par un tableau de booléens indexé par le numéro du sommet. Notez l'utilisation de la méthode `sommetsAdjacents` qui retourne l'énumération des successeurs d'un sommet.

```
private boolean[] marque;
// Parcours en largeur des successeurs du sommet s
// l'opération op est appliquée sur chaque sommet
private void pLargeur(Sommet s, Opération op) {
    File f=new FileChaînée(Sommet.class);
    marque[numéro(s)]=true;
    f.enfiler(s);
    while (!f.estVide()) {
        Sommet p=(Sommet) f.premier();
        // traiter le sommet p
        op.exécuter(p);
        f.défiler();
        // parcourir les successeurs de p
        Énumération e=sommetsAdjacents(p);
        while (!e.finÉnumération()) {
            Sommet succ=(Sommet) e.élémentSuivant();
            if (!marque[numéro(succ)]) {
                marque[numéro(s)]=true;
                f.enfiler(succ);
            }
        }
    }
}
// Parcours en largeur du graphe courant
// l'opération op est appliquée sur tous les sommets
public void parcoursLargeur(Opération op) {
    marque=new boolean[ordre()]
    Énumération g=grapheÉnumération();
```

```
    while (!g.finÉnumération()) {
        Sommet s=(Sommet) g.élémentSuivant();
        if (!marque[sommet(s)]) pLargeur(s,op);
    }
} // fin parcoursLargeur
```

Notez que l'utilisation de ces méthodes de parcours n'a de sens que si l'ordre de parcours des sommets est important. S'il s'agit d'appliquer un traitement sur chacun des sommets du graphe dans un ordre quelconque, il conviendra d'utiliser la méthode `grapheÉnumération` qui retourne l'énumération des sommets du graphe.

18.5 EXERCICES

Exercice 18.1. Déterminez le nombre maximum d'arêtes d'un graphe simple G à n sommets et p composantes connexes.

Exercice 18.2. Soit un graphe à sept sommets donné par les listes d'adjacence suivantes :

s1 → *s2* *s3* *s5*
s3 → *s4* *s6*
s4 → *s7*
s5 → *s2*
s6 → *s4*

Dessinez le graphe que ces listes représentent, puis donnez la matrice d'adjacence associée. Pour ce graphe particulier, quelle représentation vaut-il mieux choisir pour économiser de la place mémoire ?

Exercice 18.3. En partant du sommet *s1* du graphe précédent, donnez l'ordre de parcours des sommets pour les trois types de parcours :

- profondeur préfixe ;
- profondeur postfixe ;
- largeur.

Exercice 18.4. Rédigez entièrement les deux classes d'implantation de l'interface `Graphe` (donnée à la page 211) à l'aide d'une matrice d'adjacence et des listes d'adjacence.

Exercice 18.5. Implantez la notion de graphe *valué* avec les deux formes de représentation précédentes.

Exercice 18.6. Complétez les classes précédentes avec les méthodes de parcours en largeur et en profondeur.

Exercice 18.7. Écrivez une fonction qui, à partir de la représentation matricielle d'un graphe, retourne sa représentation sous forme de listes d'adjacence. Écrivez sa réciproque.

Exercice 18.8. On définit l'opération *union* qui retourne l'union de deux graphes. Sa signature est la suivante :

$$\text{union} \quad : \quad \mathcal{G}raphe \times \mathcal{G}raphe \quad \rightarrow \quad \mathcal{G}raphe$$

Donnez les axiomes qui décrivent la sémantique de cette opération et programmez une méthode `union` pour les différentes représentations de graphe.

Exercice 18.9. On appelle *puits* d'une composante connexe, un sommet s tel que pour tout sommet $x \neq s$, il existe un arc (x,s) mais pas l'arc (s,x). Montrez qu'un graphe ne peut avoir au maximum qu'un seul puits et écrivez la fonction *chercherPuits* qui recherche l'existence d'un puits dans un graphe.

Chapitre **19**

Structures arborescentes

Les structures arborescentes permettent de représenter l'information organisée de façon hiérarchique. Les arbres généalogiques, les systèmes de fichiers de la plupart des systèmes d'exploitation, ou encore le texte d'un programme informatique (voir la figure 19.1) sont parmi les nombreux exemples de structures hiérarchiques que l'on représente par des *arbres*. L'objet de ce chapitre est l'étude de la structure d'arbre qui est une des structures de données les plus importantes en informatique.

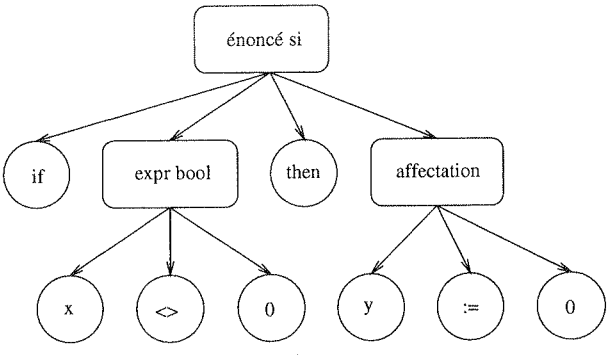


Figure 19.1 – *Un arbre syntaxique de l'énoncé Pascal* `if x<>0 then y:=0`.

Après avoir introduit la terminologie relative aux arbres, nous présenterons d'abord les arbres sous leur forme générale, puis nous en étudierons un forme particulière, les arbres binaires.

19.1 TERMINOLOGIE

Un arbre est formé d'un ensemble de sommets, appelés *nœuds*, reliés par des *arcs* et organisés de façon hiérarchique. C'est en fait un graphe connexe sans cycle. Il existe un nœud particulier appelé *racine* qui est à l'origine de l'arborescence. Contrairement aux arbres biologiques, les arbres informatiques « poussent » vers le bas, et leur racine est située au sommet de la hiérarchie. Chaque nœud possède zéro ou plusieurs *fils*, reliés avec lui de façon univoque. Chaque nœud, à l'exception de la racine, possède un *père* unique. Les fils d'un même père sont évidemment des *frères*.

Tout nœud n d'un arbre est accessible par un *chemin* unique qui part de la racine et passe par un ensemble de nœuds appelés *ascendants* de n. La racine d'un arbre est donc un ascendant de tous les nœuds de l'arbre. Réciproquement, tous les nœuds accessibles par un chemin à partir d'un nœud n sont des *descendants* de ce nœud.

La figure 19.2 page 222 montre un arbre qui possède quatorze nœuds, nommés n_1 jusqu'à n_{14}. Il est important de comprendre que les nœuds sont distincts et portent chacun un nom *unique*. Dans l'exemple, l'ordre de nomination doit être considéré comme quelconque. La racine s'appelle n_1, le nœud n_{10} est un descendant des nœuds n_1 et n_8, alors que n_2 est l'ascendant des nœuds n_4 et n_5.

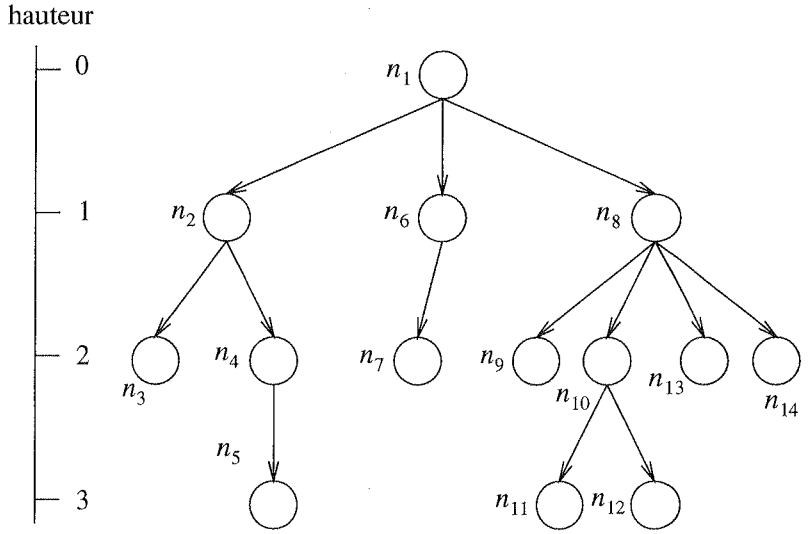


Figure 19.2 – *Un arbre formé de quatorze nœuds .*

Au chapitre 15, nous avons vu comment définir des objets récursivement, et les arbres se prêtent bien à de telles définitions. Un arbre est ainsi formé d'un nœud racine et d'une suite, éventuellement vide, d'arbres appelés *sous-arbres*. L'arbre de la figure 19.2 est formé de la racine n_1, et de trois sous-arbres. Son premier sous-arbre a pour racine le nœud n_2 et deux sous-arbres, son deuxième sous-arbre a pour racine le nœud n_6 et un unique sous-arbre, etc.

Les nœuds qui ne possèdent pas de sous-arbres, qui n'ont donc pas de fils, sont appelés *nœuds externes* ou *feuilles*. Ceux qui possèdent au moins un sous-arbre s'appellent des *nœuds*

internes. L'arbre de la figure 19.2 possèdent six nœuds internes et huit feuilles. Les nœuds n_5 et n_{14} sont des feuilles, alors que n_8 et n_{10} sont des nœuds internes.

Une *branche* d'un arbre est un chemin entre la racine et une feuille. Le nombre de branches d'un arbre est égal à son nombre de feuilles. La *longueur* d'un chemin est définie entre deux nœuds appartenant à une même branche, et est égale au nombre d'arcs qui les séparent. La longueur du chemin entre les nœuds n_8 et n_{11} est égale à deux, celle entre les points n_2 et n_4 est égale à un. Notez que la longueur d'un chemin d'un nœud unique est zéro.

La *hauteur* ou le *niveau* d'un nœud est la longueur du chemin entre la racine et lui. La hauteur ou la *profondeur* d'un arbre est égale au maximum des hauteurs de ses nœuds. La hauteur du nœud n_{10} est égale à deux, et la hauteur de l'arbre est trois. La profondeur moyenne d'un arbre est égale à la somme des hauteurs de chacun de ses nœuds divisée par le nombre de nœuds.

Un arbre *étiqueté* est un arbre dont les nœuds possèdent une valeur. Dans certains arbres, seules les feuilles sont étiquetées. Le nom du nœud et sa valeur sont deux notions distinctes, et plusieurs nœuds peuvent posséder des valeurs identiques. La figure 19.3 page 223 montre l'arbre précédent étiqueté avec des caractères alphabétiques.

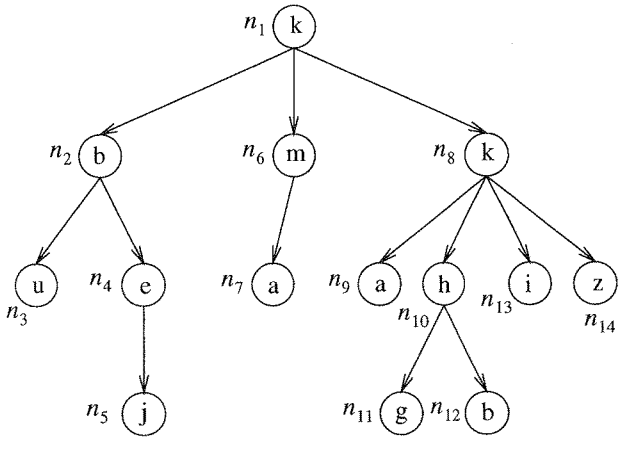


Figure 19.3 – *Un arbre étiqueté.*

19.2 LES ARBRES

Un arbre possède la forme générale que nous avons présentée plus haut. Il est composé d'un nombre fini de nœuds dont le nombre de fils est quelconque. Un arbre possède toujours au moins un nœud et n'est donc jamais vide. Plus formellement, un arbre est décrit par l'équation récursive suivante :

$$\begin{aligned} \mathcal{A}rbre &= \mathcal{N}\text{œ}ud \times \mathcal{F}or\hat{e}t \\ \mathcal{F}or\hat{e}t &= \mathcal{A}rbre^* \end{aligned}$$

Une forêt est une suite quelconque d'arbres. $\mathcal{A}rbre^*$ définit les suites d'arbres de longueur nulle, un, deux, trois, etc. et que l'on note :

$$\mathcal{A}rbre^* = \emptyset + < \mathcal{A}rbre > + < \mathcal{A}rbre\ \mathcal{A}rbre > + < \mathcal{A}rbre\ \mathcal{A}rbre\ \mathcal{A}rbre > \ldots$$

19.2.1 Définition abstraite

$\mathcal{A}rbre$ est l'ensemble des arbres, $\mathcal{N}\text{œ}ud$ l'ensemble des nœuds d'un arbre, et $\mathcal{F}or\hat{e}t$ l'ensemble des suites d'arbres. Une suite vide d'arbres est désignée par la constante *forêtvide*.

Ensembles

$\mathcal{A}rbre$ **utilise** $\mathcal{N}\text{œ}ud$, $\mathcal{F}or\hat{e}t$
$\mathcal{F}or\hat{e}t$ **utilise** $\mathcal{A}rbre$, naturel, entier
forêtvide $\in \mathcal{F}or\hat{e}t$

Pour les arbres étiquetés, nous ajouterons l'ensemble $\mathcal{E}$ des valeurs qui peuvent être associées à un nœud.

Description fonctionnelle

Les opérations données ci-dessous permettent la construction d'arbre. D'autres les compléteront par la suite.

cons	:	$\mathcal{N}\text{œ}ud \times \mathcal{F}or\hat{e}t$	$\rightarrow$	$\mathcal{A}rbre$
racine	:	$\mathcal{A}rbre$	$\rightarrow$	$\mathcal{N}\text{œ}ud$
forêt	:	$\mathcal{A}rbre$	$\rightarrow$	$\mathcal{F}or\hat{e}t$

L'opération *cons* construit un arbre à partir d'un nœud et d'une forêt. L'opération *racine* retourne le nœud de racine d'un arbre, et *forêt* retourne ses fils. Si le nœud est étiqueté, le type abstrait est complété par les opérations suivantes :

cons	:	$\mathcal{N}\text{œ}ud \times \mathcal{E}$	$\rightarrow$	$\mathcal{N}\text{œ}ud$
valeur	:	$\mathcal{N}\text{œ}ud$	$\rightarrow$	$\mathcal{E}$

La première construit un nœud étiqueté, et la seconde retourne sa valeur.

Enfin, les opérations suivantes manipulent l'ensemble $\mathcal{F}or\hat{e}t$. Elles sont semblables à celles du type abstrait $\mathcal{L}iste$, puisqu'une forêt est une suite linéaire d'arbres.

longueur	:	$\mathcal{F}or\hat{e}t$	$\rightarrow$ naturel
ièmeArbre	:	$\mathcal{F}or\hat{e}t \times$ entier	$\rightarrow \mathcal{A}rbre$
ajouterArbre	:	$\mathcal{F}or\hat{e}t \times$ entier $\times \mathcal{A}rbre$	$\rightarrow \mathcal{F}or\hat{e}t$
supprimerArbre	:	$\mathcal{F}or\hat{e}t \times$ entier	$\rightarrow \mathcal{F}or\hat{e}t$

Description axiomatique

La sémantique des opérations du type abstrait est donnée par les axiomes suivants. Notez que les axiomes de (4) à (13) sont ceux qui définissent la construction d'une liste.

$\forall a \in \mathcal{A}rbre, \forall n \in \mathcal{N}\text{œ}ud, \text{et} \forall f \in \mathcal{F}or\hat{e}t$

(1) $\text{racine}(\text{cons}(n,a)) = n$

(2) $\text{forêt}(\text{cons}(n,f)) = f$

(3) $\text{cons}(\text{racine}(a), \text{forêt}(a)) = a$

(4) $\text{longueur}(\text{forêtvide}) = 0$

(5) $\forall k, 1 \leqslant k \leqslant \text{longueur}(f)$,
$\text{longueur}(\text{supprimerArbre}(f,k)) = \text{longueur}(f) - 1$

(6) $\forall k, 1 \leqslant k \leqslant \text{longueur}(f) + 1$,
$\text{longueur}(\text{ajouterArbre}(f,k,a)) = \text{longueur}(f) + 1$

(7) $\forall k, 1 \leqslant k \leqslant \text{longueur}(f) \text{ et } 1 \leqslant i < k$,
$\text{ièmeArbre}(\text{supprimerArbre}(f,k),i) = \text{ièmeArbre}(f,i)$

(8) $\forall k, 1 \leqslant k \leqslant \text{longueur}(f) \text{ et } k \leqslant i \leqslant \text{longueur}(f) - 1$,
$\text{ièmeArbre}(\text{supprimerArbre}(f,k),i) = \text{ièmeArbre}(f,i+1)$

(9) $\forall k, 1 \leqslant k \leqslant \text{longueur}(f) + 1 \text{ et } 1 \leqslant i < k$,
$\text{ièmeArbre}(\text{ajouterArbre}(f,k,a),i) = \text{ièmeArbre}(f,i)$

(10) $\forall k, 1 \leqslant k \leqslant \text{longueur}(f) + 1 \text{ et } k = i$,
$\text{ièmeArbre}(\text{ajouterArbre}(f,k,a),i) = a$

(11) $\forall k, 1 \leqslant k \leqslant \text{longueur}(f) + 1 \text{ et } k < i \leqslant \text{longueur}(f) + 1$,
$\text{ièmeArbre}(\text{ajouterArbre}(f,k,a),i) = \text{ièmeArbre}(f,i-1)$

(12) $\forall r, r < 1 \text{ et } r > \text{longueur}(f), \nexists f', f' = \text{supprimerArbre}(f,r)$

(13) $\forall r, r < 1 \text{ et } r > \text{longueur}(f) + 1, \nexists f', f' = \text{ajouterArbre}(f,r,a)$

Enfin, si l'arbre est étiqueté, on ajoute l'axiome :

(14) $\text{valeur}(\text{cons}(n,e)) = e$

19.2.2 L'implantation en Java

Les signatures des opérations du type abstrait *Arbre* sont décrites par l'interface JAVA suivante :

```
public interface Arbre {
    public Noeud racine();
    public Forêt forêt();
}
```

L'opération *cons* du type abstrait sera définie par le constructeur de la classe qui implantera cette interface.

L'interface suivante définit le type *Forêt*. Notez la présence de la méthode `lesFrères` qui retourne l'énumération des sous-arbres de la forêt courante.

```
public interface Forêt {
  public int longueur();
  public Arbre ièmeArbre(int r) throws RangInvalideException;
  public void ajouterArbre(int r, Arbre a) throws RangInvalideException;
  public void supprimerArbre(int r) throws RangInvalideException;
  public Énumération lesFrères();
}
```

Dans ce qui suit, nous décrivons deux organisations de la structure d'arbre. La première utilise une structure chaînée, la seconde des listes d'adjacence.

Utilisation d'une structure chaînée

Une première mise en œuvre possible des arbres est une représentation chaînée des nœuds. Un nœud est un objet qui contient une forêt, implantée par une liste d'arbres. Si l'arbre est étiqueté, le nœud possède en plus une valeur. Un nœud de l'arbre est donné par la déclaration :

```
public class Noeud {
    protected Object valeur; // la valeur du nœud courant
    protected Forêt laForêt; // et ses fils
    public Noeud(Object e, Forêt f) {
        valeur=e; laForêt=f;
    }
    public Forêt forêt() { return laForêt; }
    public Object valeur() { return valeur; }
}
```

Un arbre dont les nœuds sont chaînés est donné par la classe `ArbreChaîné` qui étend la classe `Noeud` et implante l'interface `Arbre`.

```
public class ArbreChaîné extends Noeud implements Arbre
{
    public ArbreChaîné(Object e, Forêt f) {
        super(e,f);
    }
    public Forêt forêt() {
        return super.forêt();
    }
    public Noeud racine() {
        return this;
    }
}
```

La forêt est définie comme une liste d'arbres. La classe qui la décrit hérite d'une implantation particulière du type abstrait *Liste*. Le choix de cette implantation est fonction du type d'arbre à représenter. Si le nombre de fils est à peu près constant par nœud, on choisira un tableau, alors que dans le cas contraire, une structure dynamique chaînée conviendra mieux. Cette dernière organisation est souvent appelée représentation *fils-aîné-fils-droit* (voir la figure 19.4 page 227). Notez qu'elle minimise le nombre de références, mais qu'elle perd l'accès en $\mathcal{O}(1)$ aux fils.

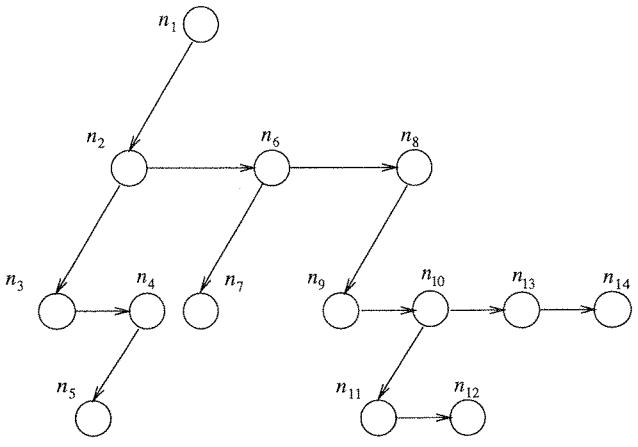


Figure 19.4 – *Représentation* fils-aîné-fils-droit *de l'arbre de la figure 19.2.*

```
public class ForêtChaînée extends ListeChaînée implements Forêt
{
    public ForêtChaînée() throws Exception {
        super(ArbreChaîné.class);
    }
    public Arbre ièmeArbre(int r) throws RangInvalideException {
        return (Arbre) super.ième(r);
    }
    public void ajouterArbre(int r, Arbre a)
    throws RangInvalideException
    {
        super.ajouter(r,a);
    }
    public void supprimerArbre(int r) throws RangInvalideException {
        super.supprimer(r);
    }
    public Énumération lesFrères() {
        return super.listeÉnumération();
    }
}
```

➢ Utilisation des listes d'adjacence

Une autre façon de représenter les arbres est d'utiliser des listes d'adjacence semblables à celles employées pour implanter des graphes. On construit une suite linéaire de tous les nœuds et on associe à chacun des nœuds la liste de ses fils. La figure 19.5 donne l'arbre de la figure 19.2 selon cette organisation.

Si la suite est implantée par un tableau, cette représentation offre un accès en $\mathcal{O}(1)$ à chaque nœud de façon indépendante de sa position dans la hiérarchie, mais la gestion dynamique des ajouts et des suppressions des nœuds dans l'arbre est plus difficile.

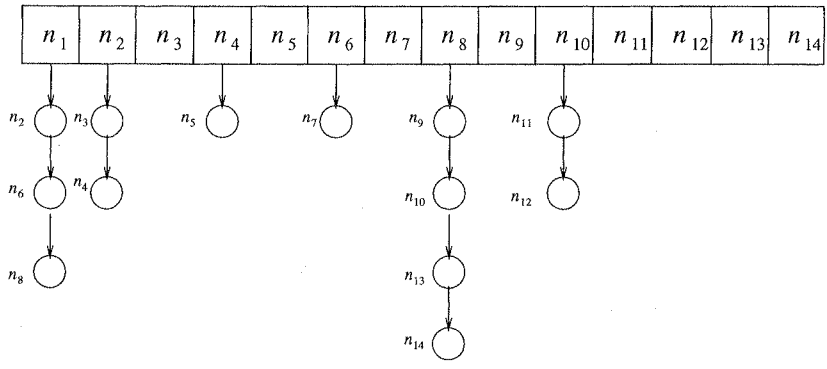


Figure 19.5 – *Arbre représenté par des listes d'adjacence.*

19.2.3 Algorithmes de parcours d'un arbre

Comme pour un graphe, le parcours d'un arbre passe par tous les nœuds pour appliquer un traitement, toujours le même, sur chacun d'entre eux. Deux types de parcours sont possibles : en profondeur et en largeur. Nous présentons l'algorithme de parcours en profondeur, celui du parcours en largeur est laissé en exercice.

Le parcours en profondeur d'un arbre a consiste à passer par sa racine, puis à parcourir en profondeur chacun de ses fils. L'algorithme s'exprime récursivement comme suit :

```
Algorithme Parcours-en-Profondeur(a)
    {Parcours en profondeur de l'arbre a}
    pourtout fils de forêt(a) faire
        {parcourir en profondeur le fils courant}
        Parcours-en-Profondeur(fils)
    finpour
```

Lors du parcours de l'arbre, si le traitement est appliqué sur la racine avant l'énoncé itératif, le parcours est *préfixe*. Au contraire, s'il a lieu après, le parcours est *postfixe*.

Nous donnons ci-dessous la programmation en JAVA de la méthode de parcours préfixe de la classe `Arbre`. Le traitement des nœuds de l'arbre est assuré par une opération `exécuter` du paramètre `op` de type `Opération` (voir la section 18.4.3, page 218).

```
public void parcoursPréfixe(Opération op) {
    op.exécuter(valeur());
    Énumération f=forêt().lesFrères();
    while (!f.finÉnumération())
        ((Arbre) f.élémentSuivant()).parcoursPréfixe(op);
}
```

19.3 ARBRE BINAIRE

Un arbre binaire est un arbre qui possède au plus deux fils, un *sous-arbre gauche* et un *sous-arbre droit*. Les arbres binaires sont utilisés dans de nombreuses circonstances. Ils servent, par exemple, à représenter des généalogies ou des expressions arithmétiques (voir la figure 19.6).

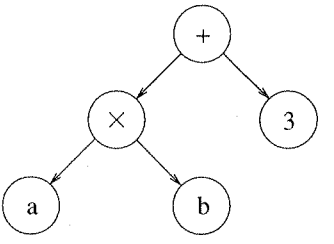


Figure 19.6 – *L'expression a × b + 3.*

Un arbre binaire n'est toutefois pas un arbre dont la forêt serait limitée à deux fils. Les deux arbres donnés par la figure 19.7 sont différents et ne peuvent pas être distingués avec un arbre à un seul fils. Le premier possède un fils gauche et pas de fils droit. Inversement, le second possède un fils droit et pas de fils gauche.

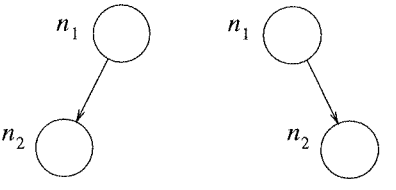


Figure 19.7 – *Deux arbres binaires distincts.*

L'arbre binaire (a) de la figure 19.8 possède une forme quelconque, mais certains arbres ont des formes caractéristiques. On appelle arbre binaire *dégénéré* (b), un arbre dont chaque niveau possède un seul nœud (les nœuds appartiennent à une seule et même branche). Un arbre binaire *complet* (c) est un arbre dont les nœuds, qui ne sont pas des feuilles, possèdent toujours deux fils. Enfin, un arbre binaire *parfait* (d) est un arbre dont toutes les feuilles sont situées sur au plus deux niveaux ; les feuilles du dernier niveau sont placées le plus à gauche.

Un arbre binaire de n nœuds possède une profondeur p minimale lorsqu'il est parfait et maximale lorsqu'il est dégénéré. La profondeur p et le nombre de nœuds n d'un arbre sont tels que $\lfloor \log_2 n \rfloor \leqslant p \leqslant n - 1$, où $\lfloor \rfloor$ désigne la partie entière inférieure. Cette relation est très importante car elle détermine la complexité de la plupart des algorithmes sur les arbres binaires. Cette complexité est comprise entre $\mathcal{O}(\log_2 n)$ et $\mathcal{O}(n)$.

Une autre relation intéressante lie le nombre de feuilles et le nombre de nœuds. Pour tout arbre binaire, le nombre de feuilles est égal au nombre de nœuds plus 1.

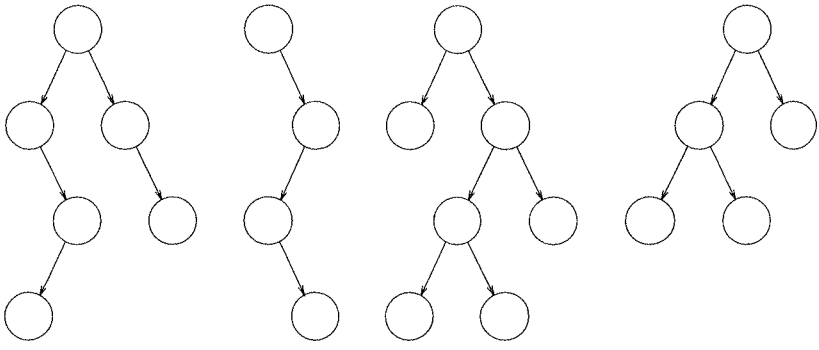


Figure 19.8 – *Arbre (a) quelconque (b) dégénéré (c) complet (d) parfait.*

19.3.1 Définition abstraite

Comme pour les arbres dont le nombre de sous-arbres est quelconque, nous pouvons donner une définition récursive d'un arbre binaire. Les équations qui décrivent un arbre binaire sont les suivantes :

$$\begin{aligned} \mathcal{A}rbre_b &= \emptyset \\ \mathcal{A}rbre_b &= \mathcal{N}\text{œ}ud \times \mathcal{A}rbre_b \times \mathcal{A}rbre_b \end{aligned}$$

Elles signifient qu'un arbre binaire est soit vide, soit formé d'un nœud et de deux arbres binaires, appelés respectivement sous-arbre gauche et sous-arbre droit. Notez que la notion d'arbre binaire vide, étrangère aux arbres, a été introduite pour distinguer les deux arbres binaires de la figure 19.7 page 229.

➤ Ensembles

$\mathcal{A}rbre_b$ est l'ensemble des arbres binaires et possède l'élément particulier arbrevide qui correspond à un arbre binaire vide. Les nœuds de l'arbre appartiennent à l'ensemble $\mathcal{N}\text{œ}ud$.

$\mathcal{A}rbre_b$ **utilise** $\mathcal{N}\text{œ}ud$ et booléen
arbrevide $\in \mathcal{A}rbre_b$

➤ Description fonctionnelle

Les opérations suivantes sont définies sur le type $\mathcal{A}rbre_b$:

$$\begin{array}{lcll} \text{cons} & : & \mathcal{N}\text{œ}ud \times \mathcal{A}rbre_b \times \mathcal{A}rbre_b & \rightarrow \mathcal{A}rbre_b \\ \text{racine} & : & \mathcal{A}rbre_b & \rightarrow \mathcal{N}\text{œ}ud \\ \text{sag} & : & \mathcal{A}rbre_b & \rightarrow \mathcal{A}rbre_b \\ \text{sad} & : & \mathcal{A}rbre_b & \rightarrow \mathcal{A}rbre_b \\ \text{est-vide?} & : & \mathcal{A}rbre_b & \rightarrow \text{booléen} \end{array}$$

Comme pour les arbres généraux, les opérations suivantes sont définies sur les nœuds étiquetés :

$$\begin{array}{lcll} \text{cons} & : & \mathcal{N}\text{œud} \times \mathcal{E} & \rightarrow \mathcal{N}\text{œud} \\ \text{valeur} & : & \mathcal{N}\text{œud} & \rightarrow \mathcal{E} \end{array}$$

Description axiomatique

Les axiomes suivants décrivent la sémantique des opérations du type abstrait $\mathcal{A}rbre_b$. Les deux premiers axiomes spécifient un arbre vide, le troisième la façon de construire un arbre binaire, et les derniers l'accès et les conditions d'accès aux composants d'un arbre binaire.

$\forall n \in \mathcal{N}\text{œud}, \forall\, a,g,d \in \mathcal{A}rbre_b$

(1) est-vide?(arbrevide) = vrai

(2) est-vide?(cons(n,g,d)) = faux

(3) cons(racine(a), sag(a), sad(a)) = a

(4) racine(cons(n,g,d)) = n

(5) sag(cons(n,g,d)) = g

(6) sad(cons(n,g,d)) = d

(7) $\nexists n \in \mathcal{E}$, n = racine(arbrevide)

(8) $\nexists a \in \mathcal{A}rbre_b$, a = sag(arbrevide)

(9) $\nexists a \in \mathcal{A}rbre_b$, a = sad(arbrevide)

L'axiome suivant est défini pour un arbre binaire étiqueté :

$\forall n \in \mathcal{N}\text{œud}$ et $\forall e \in \mathcal{E}$

(10) valeur(cons(n,e)) = e

19.3.2 L'implantation en Java

L'interface `ArbreBinaire` donne les signatures des opérations du type abstrait $\mathcal{A}rbre_b$. L'opération *cons* sera donnée par le constructeur des classes qui implanteront cette interface.

```
public interface ArbreBinaire {
    public NoeudBinaire racine() throws ArbreVideException;
    public ArbreBinaire sag() throws ArbreVideException;
    public ArbreBinaire sad() throws ArbreVideException;
    public boolean estVide();
}
```

Les arbres binaires sont généralement utilisés pour la mise en œuvre de structures dynamiques et sont implantés par des structures chaînées. Toutefois, dans le cas très particulier des arbres parfaits, on choisit souvent une implantation avec des tableaux.

Structures chaînées

Avec cette organisation, les nœuds de l'arbre binaire sont reliés par leurs sous-arbres gauche ou droit. Un nœud porte deux références à ses sous-arbres, plus une valeur s'il est étiqueté. Un nœud est représenté par la classe ci-dessous.

```
public class NoeudBinaire {
    protected Object valeur; // la valeur du nœud courant
    protected ArbreBinaire sag, sad; // et ses fils gauche et droit
    public NoeudBinaire(Object e, ArbreBinaire g, ArbreBinaire d)
    {
        valeur=e;
        sag=g;
        sad=d;
    }
    public NoeudBinaire(Object e) {
        valeur=e;
        sag=sad=ArbreBinaireChaîné.arbreVide;
    }
    public Object valeur() { return valeur; }
    public void changerValeur(Object o) { valeur=o; }
}
```

Un arbre binaire est simplement une référence sur un nœud. La définition d'un arbre binaire dont les nœuds sont chaînés est alors obtenue par héritage de la classe `NoeudBinaire`. Un arbre vide est représenté par un nœud particulier, désigné par la constante de classe `arbreVide`. Ce choix, plutôt que celui de la valeur **`null`**, est conditionnée par la méthode `estVide`. Si la valeur **`null`** est utilisée, un objet de type arbre binaire ne pourra jamais tester s'il est vide dans la mesure où l'objet doit exister pour que la méthode puisse être exécutée ; il sera donc toujours différent de **`null`**. Le coût supplémentaire en espace mémoire est celui d'une *seule* constante `arbreVide` pour tout arbre binaire. L'opération *cons* du type abstrait $Arbre_b$ est définie par le constructeur de la classe.

```
public class ArbreBinaireChaîné extends NoeudBinaire
implements ArbreBinaire
{
   public static final ArbreBinaire arbreVide =
                                          new ArbreBinaireChaîné(null);
   public ArbreBinaireChaîné(Object e, ArbreBinaire g, ArbreBinaire d) {
       super(e,g,d);
   }
   public ArbreBinaireChaîné(Object e) {
       super(e);
   }
   public boolean estVide() {
       return this==arbreVide;
   }
   public NoeudBinaire racine() throws ArbreVideException {
       if (this==arbreVide)
           throw new ArbreVideException();
       return this;
   }
```

```
    public ArbreBinaire sag() throws ArbreVideException {
        if (this==arbreVide)
            throw new ArbreVideException();
        return sag;
    }
    public ArbreBinaire sad() throws ArbreVideException {
        if (this==arbreVide)
            throw new ArbreVideException();
        return sad;
    }
} // fin classe ArbreBinaireChaîné
```

Utilisation d'un tableau

L'utilisation d'un tableau est adaptée aux arbres qui évoluent peu, et plus particulièrement aux arbres parfaits. Considérons l'arbre parfait de la figure 19.9.

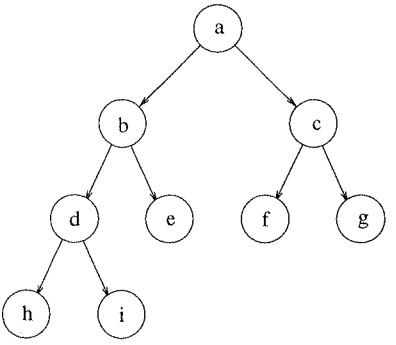


Figure 19.9 – *Un arbre parfait à neuf nœuds .*

Les éléments de cet arbre sont rangés dans le tableau par niveau, comme le montre la figure suivante :

1	2	3	4	5	6	7	8	9
a	b	c	d	e	f	g	h	i

Avec une telle organisation, un nœud d'indice i, avec $1 \leqslant i \leqslant n$ div 2, possède un sous-arbre gauche à l'indice $2i$ et un sous-arbre droit à l'indice $2i + 1$. Inversement, le père d'un nœud d'indice i, avec $2 \leqslant i \leqslant n$, est à l'indice en i div 2.

Cette représentation peut servir pour un arbre binaire quelconque, mais convient plus particulièrement aux arbres parfaits car tous les composants du tableau sont utilisés. Au contraire, cette représentation sera évidemment à exclure pour un arbre dégénéré. Un tel arbre qui possède n nœuds peut nécessiter un tableau de $2^n - 1$ composants. Nous verrons à la section 22.2.2 une illustration de cette représentation des arbres parfaits avec la technique de tri en tas.

19.3.3 Parcours d'un arbre binaire

Parcours en profondeur

Le parcours en profondeur consiste à passer par le nœud courant, et à parcourir en profondeur le sous-arbre gauche, puis le sous-arbre droit.

```
Algorithme Parcours-en-Profondeur(a)
    {Parcours en profondeur de l'arbre binaire a}
    si non estvide(a) alors
        Parcours-en-Profondeur(sag(a))
        Parcours-en-Profondeur(sad(a))
    finsi
```

Selon le moment où le nœud courant est traité, on distingue trois types de parcours en profondeur, *préfixe*, *infixe* et *postfixe*. Le parcours *préfixe* traite en premier le nœud, puis parcourt les deux sous-arbres. Le parcours *infixe* parcourt le sous-arbre gauche, traite le nœud, et parcourt le sous-arbre droit. Enfin, Le parcours *postfixe* parcourt d'abord les deux sous-arbres, et traite le nœud en dernier.

La programmation en JAVA du parcours infixe d'un arbre binaire est donnée ci-dessous. Cette méthode complète la classe `ArbreBinaire`. Le traitement à appliquer à chaque nœud est donné par le paramètre `op` de type `Opération` (voir à la page 218).

```
// Parcours en profondeur de l'arbre binaire courant
// L'opération op est appliquée sur chacun de ses noeuds
public void parcoursInfixe(Opération op) {
    if (!estVide()) {
        sag().parcoursInfixe(op);
        op.exécuter(racine().valeur());
        sad().parcoursInfixe(op);
    }
}
```

Parcours en largeur

L'affichage vertical d'un arbre binaire sur une imprimante ou un terminal qui écrit ligne par ligne impose un parcours en largeur de l'arbre. Comme pour l'algorithme de parcours en largeur d'un graphe (voir à la page 216), une file d'attente est nécessaire pour conserver les nœuds traités à chaque niveau. Toutefois, l'algorithme de parcours de l'arbre est plus simple que celui du graphe, puisque l'arbre étant par définition connexe et sans cycle, il est inutile de gérer des marques pour s'assurer qu'un nœud n'a pas déjà été traité.

```
Algorithme Parcours-en-Largeur(a)
    {Parcours en largeur de l'arbre binaire a}
    si non est-vide(a) alors
        enfiler(f, a)
        répéter
            b ← premier(f)
            défiler(f)
            traiter(racine(b))
            {enfiler les sous-arbres d'un même niveau}
```

```
            si non est-vide(sag(b)) alors
               enfiler(f,sag(b))
            finsi
            si non est-vide(sad(b)) alors
               enfiler(f,sad(b))
            finsi
         jusqu'à est-vide(f)
      finsi
```

La programmation de cet algorithme est donnée par la méthode `parcoursEnLargeur` suivante qui complète la classe `ArbreBinaire`.

```
public void parcoursEnLargeur(Opération op) {
   if (!estVide()) {
       File f=new FileChaînée(ArbreBinaireChaîné.class);
       f.enfiler(this);
       do {
           ArbreBinaire b=(ArbreBinaire) f.premier();
           op.exécuter(b.racine().valeur());
           f.défiler();
          // enfiler les sous-arbres d'un même niveau
           if (!b.sag().estVide()) f.enfiler(b.sag());
           if (!b.sad().estVide()) f.enfiler(b.sad());
       } while (!f.estVide());
   }
}
```

19.4 REPRÉSENTATION BINAIRE DES ARBRES GÉNÉRAUX

Tout arbre peut être représenté par un arbre binaire. La représentation d'un arbre a par un arbre binaire b est donnée par les règles de transformation suivantes :

1. racine(b) = racine(a) ;
2. tous les transformés binaires des fils de a sont liés entre eux leur sous-arbre droit ;
3. le sous-arbre gauche de b est le transformé binaire du premier fils de a.

La figure suivante montre la transformation d'un arbre qui possède une racine et une forêt de k sous-arbres (à gauche) en son équivalent binaire (à droite).

L'algorithme qui transforme un arbre général en un arbre binaire s'exprime récursivement. Les transformés binaires des sous-arbres d'une forêt sont liés par leur sous-arbre droit en partant du dernier sous-arbre de la forêt, puis en remontant jusqu'au premier sous-arbre. Le sous-arbre gauche du nœud courant est lié au premier sous-arbre transformé sous forme binaire.

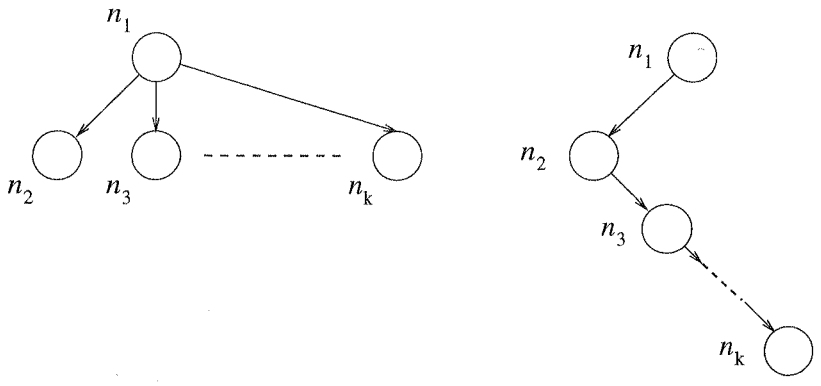


Figure 19.10 – *Transformation d'un arbre général en arbre binaire.*

```
Algorithme Transformé-Binaire(a, b)
   {Rôle : transforme l'arbre a en arbre binaire b}
       racine(b) ← racine(a)
       sad(b) ← arbrevide
       {lier les sous-arbres droits des transformés binaires}
       {des arbres de la forêt de a}
       aîné ← arbrevide
       pourtout i de longueur(forêt(a)) à 1 faire
            Transformé-Binaire(ièmeArbre(forêt(a),i), f)
            sad(f) ← aîné
            aîné ← f
       finpour
       sag(b) ← aîné
```

Nous donnons ci-dessous la programmation en JAVA de cet algorithme. Notez que la méthode `changerSad` a été ajoutée à la classe `Noeud`. Elle permet de changer le sous-arbre droit de l'arbre binaire courant.

```
public ArbreBinaire transforméBinaire() {
   ArbreBinaire frère, aîné= ArbreBinaireChaîné.arbreVide;
   for(int r=forêt().longueur(); r >= 1; r--) {
      frère=((Arbre) forêt().ièmeArbre(r)).transforméBinaire();
      frère.changerSad(aîné);
      aîné=frère;
   }
   return new ArbreBinaireChaîné(racine().valeur(),
                                 aîné, ArbreBinaireChaîné.arbreVide);
}
```

19.5 EXERCICES

Exercice 19.1. Donnez l'algorithme du parcours en largeur d'un arbre général.

Exercice 19.2. Montrez par récurrence qu'un arbre binaire de profondeur p possède au plus 2^p nœuds.

Exercice 19.3. Donnez les algorithmes qui calculent la hauteur d'un arbre quelconque et d'un arbre binaire, puis ajoutez la méthode `hauteur` aux classes `Arbre` et `ArbreBinaire`.

Exercice 19.4. Le nombre de STRAHLER [1] est une sorte de mesure de la complexité d'un arbre binaire complet. Il est défini par la fonction S suivante :

$$S(a) = \begin{cases} 0 & \text{si } a \text{ est une feuille} \\ S(\text{sag}(a)) + 1 & \text{si } S(\text{sag}(a)) = S(\text{sad}(a)) \\ \max(S(\text{sag}(a)), S(\text{sad}(a))) & \text{sinon} \end{cases}$$

Rédigez l'algorithme qui calcule ce nombre et ajoutez la méthode `strahler` à la classe `ArbreBinaire`.

Exercice 19.5. Deux arbres binaires a et b sont *miroirs* s'ils possèdent la même racine, si le sous-arbre gauche de a est le miroir du sous-arbre droit de b et si le sous-arbre droit de a est le miroir du sous-arbre gauche de b. Écrivez l'algorithme qui permet de vérifier si deux arbres sont miroirs et ajoutez la méthode `miroir` à la classe `ArbreBinaire`.

Exercice 19.6. Écrivez la version itérative de l'algorithme de parcours en profondeur d'un arbre binaire. Vous aurez besoin d'une pile, dont la hauteur est au plus égale à la profondeur de l'arbre.

Exercice 19.7. Il est possible de dénoter de façon univoque les nœuds d'un arbre binaire par des mots formés d'une suite de 0 et de 1 obtenus en parcourant le chemin qui mène de la racine à ce nœud. Par définition, la racine est le mot vide $\varnothing$, et si un nœud est dénoté par le mot m son fils gauche est $m0$ et son fils droit $m1$. Par exemple, les noeuds de l'arbre donné par la figure 19.6 de la page 229 sont tels que $+ = \varnothing$, $\times = 0$, $a = 00$, $b = 01$ et $3 = 1$.

À l'aide de cette notation, donnez :

- la représentation des nœuds des bords gauche et droit d'un arbre binaire ;
- la représentation du père d'un nœud ;
- la hauteur d'un arbre binaire.

Exercice 19.8. Soit un ensemble E d'éléments $\{x_1, x_2, \ldots, x_n\}$ munies de probabilités $p_1, p_2, \ldots, p_n$, avec $\sum_{k=1}^{n} p_k = 1$. On associe à l'ensemble E un arbre binaire construit de la façon suivante : à partir des n feuilles de l'arbre constituées par les éléments x_k, on choisit les deux éléments x_i et x_j qui possèdent les probabilités les plus petites et on construit un nouveau nœud x' ayant x_i et x_j pour fils et dont la probabilité est $p_i + p_j$ (on placera l'élément de probabilité la plus petite à gauche). Dans l'ensemble E, on remplace x_i et x_j par x'. Le

1. Utilisé à l'origine en hydrographie pour décrire la structure d'un réseau de rivières, ce nombre est utilisé en informatique par les compilateurs dans la gestion de l'allocation des registres, ou encore par des outils graphiques de visualisation de graphe.

nouvel ensemble E n'a plus que $n-1$ éléments. On recommence l'opération jusqu'à obtenir un ensemble réduit à un élément.

Dessinez l'arbre associé à l'ensemble $E = \{b,e,m,x,z\}$ munie des probabilités $p(b) = 0{,}21$, $p(e) = 0{,}35$, $p(m) = 0{,}26$, $p(x) = 0{,}08$ et $p(z) = 0{,}1$.

Pour un ensemble E de n éléments, combien de nœuds internes et de feuilles possèdera l'arbre ?

Rédigez l'algorithme qui construit l'arbre associé à un ensemble E selon la méthode précédente.

Exercice 19.9. L'arbre de l'exercice précédent s'appelle un arbre de HUFFMAN. Associé à la notation présentée à l'exercice n° 19.7, cet arbre permet de définir un code binaire unique pour les éléments d'un ensemble E. Pour l'ensemble E précédent, on obtient le code : $b = 11$, $e = 01$, $m = 00$, $x = 100$ et $z = 101$. Avec un tel code, la suite 00011101101 se décode sans ambiguïté *mebez* (on décode en partant de la racine de l'arbre de HUFFMAN, et en suivant le chemin indiqué jusqu'à une feuille ; on écrit la lettre correspondante et on repart de la racine pour décoder le reste).

Le codage de HUFFMAN est utilisé pour comprimer des fichiers de caractères. Les taux de compression peuvent varier de 30% à 60% selon les fichiers. L'idée est de permettre un codage de longueur variable des caractères, avec le codage le plus court pour les lettres les plus fréquentes. Notez que pour un ensemble d'une centaine de caractères, il faut au plus $\lceil \log_2 100 \rceil = 7$ bits pour coder un caractère.

Écrivez en JAVA une classe `Huffman` qui fournit deux méthodes qui, respectivement, code et décode un fichier de texte. Au préalable, vous constituerez une table des fréquences de caractère à partir de fichiers de texte dont vous disposez.

Exercice 19.10. Un arbre étiqueté est représenté sur un fichier de texte sous la forme suivante : chaque nœud est représenté par son étiquette (une suite de lettres), suivi d'une virgule, suivie du nombre de fils du nœud (un entier naturel), suivie de la représentation de ses fils, séparés par des virgules. Dessinez l'arbre défini par la suite de caractères :

```
pierre,3,paul,01,marie,0,claude,2,maud,00,léa,1,léo,0,charles,0,
```

Écrivez l'algorithme qui construit un arbre à partir de sa représentation textuelle lue sur un fichier. Programmez cet algorithme en JAVA.

Chapitre **20**

Tables

La conservation de l'information sous des formes diverses, que ce soit en mémoire centrale ou en mémoire secondaire, et la recherche d'informations à partir de critères spécifiques est une activité très courante en informatique. Nous appellerons *table*[1] la structure qui permet de conserver des *éléments* de nature quelconque, munie des opérations d'*ajout*, de *suppression* et de *recherche*.

L'accès à un élément se fait à partir d'une qui l'identifie. Par exemple, si une table conserve des informations sur des personnes, on pourra choisir comme clé le numéro INSEE de chaque individu. Notez toutefois, que l'unicité de la clé n'est pas une nécessité, et que plusieurs éléments distincts peuvent posséder la même clé.

La façon de représenter une table aura une grande incidence sur la complexité des algorithmes de manipulation de table. Ces algorithmes s'appuient essentiellement sur des recherches basées sur des comparaisons entre clés, et nous distinguerons par la suite les recherches *positives* lorsque la clé recherchée est présente dans la table et les recherches *négatives* lorsqu'elle est absente.

Les tables peuvent être représentées de nombreuses façons. Dans ce chapitre, nous traiterons uniquement des tables placées en mémoire centrale, et représentées par des listes et des arbres binaires, ainsi que des tables d'adressage dispersé. Mais, tout d'abord, décrivons formellement la notion de table par son type abstrait *Table*.

1. Le terme *dictionnaire* est également employé pour désigner cette structure.

20.1 DÉFINITION ABSTRAITE

20.1.1 Ensembles

Table définit l'ensemble des tables qui mémorisent des éléments de l'ensemble $\mathcal{E}$. Chacun des éléments étant muni d'une clé prise dans $Clé$.

Table **utilise** $\mathcal{E}$ et $Clé$
tablevide $\in$ *Table*

20.1.2 Description fonctionnelle

Les signatures des trois opérations de base sur les tables sont données par :

$$\begin{array}{lcll} \text{ajouter} & : & \mathit{Table} \times \mathcal{E} & \rightarrow \mathit{Table} \\ \text{supprimer} & : & \mathit{Table} \times Clé & \rightarrow \mathit{Table} \\ \text{rechercher} & : & \mathit{Table} \times Clé & \rightarrow \mathcal{E} \end{array}$$

De plus, l'opération qui permet d'obtenir la clé d'un élément à partir de sa valeur est définie par :

$$\text{clé} \ : \ \mathcal{E} \ \rightarrow \ Clé$$

20.1.3 Description axiomatique

Pour cette description axiomatique, nous complétons le type abstrait par l'opération *occurrences* qui retourne le nombre d'occurrences d'une clé dans une table.

$$\text{occurrences} \ : \ \mathit{Table} \times Clé \ \rightarrow \text{naturel}$$

(1) occurrences(tablevide ,c) $= 0$
(2) clé(e) $= c \Rightarrow$ occurrences(ajouter(t,e),c) $=$ occurrences(t,c) $+ 1$
(3) clé(e) $\neq c \Rightarrow$ occurrences(ajouter(t,e),c) $=$ occurrences(t,c)
(4) occurrences(t,c) $= 0 \Rightarrow \nexists t,\ t =$ supprimer(t,c)
(5) occurrences(t,c) $\geqslant 1 \Rightarrow$
occurrences(supprimer(t,c),c) $=$ occurrences(t,c) $- 1$
(6) $c \neq c' \Rightarrow$ occurrences(supprimer(t,c),c') $=$ occurrences(t,c')
(7) occurrences(t,c) $= 0 \Rightarrow \nexists e,\ e =$ rechercher(t,c) $\in t$
(8) occurrences(t,c) $\geqslant 1 \Rightarrow$ clé(rechercher(t,c)) $= c$

20.2 REPRÉSENTATION DES ÉLÉMENTS EN JAVA

Pour toutes les représentations des tables de ce chapitre, les éléments sont formés d'une valeur et d'une clé de type quelconque. Pour les définir, nous utiliserons la classe `Élément` suivante :

```
class Élément {
    protected Object valeur, clé;
    public Élément(Object e, Object c) {
        valeur=e;
        clé=c;
    }
    public Object clé() { return clé; }
    public Object valeur() { return valeur; }
}
```

Les clés sont des objets quelconques, mais qui doivent posséder des opérateurs relationnels nécessaires aux opérations du type abstrait *Table*. À chaque liste ordonnée[2], nous associerons les opérations de comparaison propres à l'ensemble des clés utilisé. Les signatures de ces opérations sont données par l'interface suivante :

```
public interface Comparable {
    public boolean comparable(Object x);
    public boolean égal(Object x, Object y);
    public boolean inférieur(Object x, Object y);
    public boolean inférieurOuÉgal(Object x, Object y);
    public boolean supérieur(Object x, Object y);
    public boolean supérieurOuÉgal(Object x, Object y);
}
```

La méthode `comparable` vérifie si deux clés peuvent être comparées, c'est-à-dire si elles sont de même nature. Les autres méthodes se passent de commentaires. Par exemple, des clés représentées par des entiers seront implantées comme suit :

```
public class ComparateurDeCléEntière implements Comparable {
    public boolean comparable(Object x) {
        return x==null  ? false :
                Integer.class.isAssignableFrom(x.getClass());
    }
    public boolean égal(Object x, Object y) {
        return ((Integer) x).compareTo((Integer) y)==0;
    }
    public boolean inférieur(Object x, Object y) {
        return ((Integer) x).compareTo((Integer) y)<0;
    }
    public boolean inférieurOuÉgal(Object x, Object y) {
        return égal(x,y) || inférieur(x,y);
    }
```

2. Notez qu'il aurait été aussi possible de munir chaque clé de ses opérations de comparaison.

```
    public boolean supérieur(Object x, Object y) {
        return ((Integer) x).compareTo((Integer) y)>0;
    }
    public boolean supérieurOuÉgal(Object x, Object y) {
        return égal(x,y) || supérieur(x,y);
    }
} // fin classe ComparateurDeCléEntière
```

Les axiomes 4 et 7 du type abstrait *Table* montrent que les opérations *rechercher* ou *supprimer* échouent si la clé n'est pas présente dans la table. Il est possible de traiter cette situation de plusieurs façons, soit en signalant une erreur, soit en retournant un élément spécial, ou encore en ajoutant aux opérations un booléen qui indique l'échec de l'opération. Par la suite, nous retiendrons la première solution, et les opérations émettront l'exception `CléNonTrouvéeException`.

Les représentations de table que nous allons étudier maintenant, c'est-à-dire à l'aide de listes, d'arbres et de fonctions d'adressage dispersé, implantent toutes l'interface `Table` suivante :

```
public interface Table {
  public void ajouter(Élément e);
  public void supprimer(Object clé) throws CléNonTrouvéeException;
  public Élément rechercher(Object clé) throws CléNonTrouvéeException;
}
```

20.3 REPRÉSENTATION PAR UNE LISTE

20.3.1 Liste non ordonnée

La représentation d'une table par une liste non ordonnée est la méthode la plus simple et la plus naïve. L'ajout des éléments peut se faire n'importe où, et en particulier en tête ou en queue de liste, selon la représentation choisie de la liste, afin de garder une complexité en $\mathcal{O}(1)$. L'opération de suppression nécessite une recherche de l'élément à supprimer, suivie ou non d'un décalage d'éléments si la table est représentée par un tableau. L'algorithme de recherche consiste à comparer les éléments un à un jusqu'à ce que l'on ait trouvé l'élément, ou alors atteint la fin de la liste.

Une liste linéaire l peut être définie récursivement (cf. exercice 17.6) comme étant, soit la liste vide, soit la concaténation d'un élément de tête e avec une liste l' que nous noterons $< e, l' >$. La définition axiomatique de *rechercher* s'exprime alors comme suit :

(1) $\nexists e,\ e = \text{rechercher(listevide ,}c\text{)}$
(2) $\text{clé}(e) = c \Rightarrow \text{rechercher}(< e, l > ,c) = e$
(3) $\text{clé}(e) \neq c \Rightarrow \text{rechercher}(< e, l > ,c) = \text{rechercher}(l,c)$

Algorithme `rechercher(t, c)`

```
{Rôle : rechercher dans la table t l'élément de clé c}
{Conséquent : retourne l'élément e de clé c
              ∀k, 1≤k≤longueur(t), clé(ième(t,k))≠c}
```

```
i ← 1
tantque i≠longueur(t) faire
  {i≠longueur(t) et ∀k, 1≤k<i, clé(ième(t,k))≠c}
   x ← ième(t,i)
   si clé(x)=c alors
      rendre x
     sinon i ← i+1
   finsi
fintantque
{∀k, 1≤k≤longueur(t), clé(ième(t,k))≠c}
```

S'il y a équiprobabilité dans la recherche des éléments, le coût moyen d'une recherche positive, exprimé en nombre de comparaisons, est $\frac{1}{2}(n+1)$, et pour une recherche négative n.

Cet algorithme peut être légèrement amélioré grâce à une *sentinelle*. On supprime le test de fin de liste en ajoutant systématiquement la clé recherchée à l'extrémité de la liste :

```
Algorithme rechercher(t, c)
   i ← 1
   ajouter un élément bidon de clé c en fin de liste
   tantque c≠ième(t,i) faire
     {i≤longueur(t)+1 et ∀k, 1≤k<i, clé(ième(t,k))≠c}
     i ← i+1
   fintantque
   {clé(ième(t,i))=c}
   si i≤longueur(t) alors rendre ième(t,i) finsi
   {∀k, 1≤ k≤longueur(t), clé(ième(t,k))≠c}
```

La classe `ListeNonOrdonnéeChaînée` représente une liste dont les éléments sont ordonnés par des clés dont les opérateurs de comparaisons sont contenus dans l'attribut `comp` de type `Comparable`. Cette classe implante l'interface `Table`. Son constructeur mémorise les opérations de comparaison du type de clé utilisé.

```
public class ListeNonOrdonnéeChaînée extends ListeChaînée
implements Table
{
    protected Comparable comp;
    public ListeNonOrdonnéeChaînée(Comparable cmp) {
        super(Élément.class);
        comp=cmp;
    }
}
```

En suivant l'algorithme précédent, la méthode `rechercher` s'écrit :

```
public Élément rechercher(Object clé)
throws CléNonTrouvéeException
{
    Énumération énum=listeÉnumération();
    while (! énum.finÉnumération()) {
```

```
            Object x=énum.élémentSuivant();
            if (comp.égal(((Élément) x).clé(), clé))
                return (Élément) x;
        }
        // ∀k, 1≤k≤longueur(this), clé(ième(this,k))≠c
        throw new CléNonTrouvéeException();
}
```

S'il n'y a pas équiprobabilité dans la recherche des éléments, il est possible d'ordonner les éléments de façon à placer en tête de liste les éléments les plus recherchés. Mais on n'a pas toujours connaissance des éléments les plus recherchés. Dans ce cas, il faut faire évoluer la liste pour que les éléments les plus recherchés soient situés en tête. C'est ce que l'on appelle la recherche *auto-adaptative*. Citons deux algorithmes peu coûteux :

- après chaque recherche, on place l'élément recherché en tête de liste ; cette méthode est bien adaptée si la liste est représentée sous forme chaînée ;
- après chaque recherche, on fait progresser l'élément recherché d'une place vers la tête de la liste.

20.3.2 Liste ordonnée

Nous considérerons que les éléments d'une liste ordonnée l sont en ordre croissant de telle façon que :

$$\forall i,j \in [1, \text{longueur}(l)],\ i < j \Rightarrow \text{clé}(\text{ième}(l,i))) \leqslant \text{clé}(\text{ième}(l,j)))$$

Les axiomes qui définissent les opérations sont exprimées à l'aide de la définition récursive des listes.

(1) $\nexists e,\ e = \text{rechercher}(\text{listevide}, c)$
(2) $\text{clé}(e) = c \Rightarrow \text{rechercher}(< e, l >, c) = e$
(3) $\text{clé}(e) < c \Rightarrow \text{rechercher}(< e, l >, c) = \text{rechercher}(l, c)$
(4) $\text{clé}(e) > c \Rightarrow \nexists e'\ e' = \text{rechercher}(< e, l >, c)$
(5) $\text{ajouter}(\text{listevide}, e) =< e, \text{listevide} >$
(6) $\text{clé}(e) < \text{clé}(e') \Rightarrow \text{ajouter}(< e, l >, e') =< e, \text{ajouter}(l, e') >$
(7) $\text{clé}(e) \geqslant \text{clé}(e') \Rightarrow \text{ajouter}(< e, l >, e') =< e',\ < e, l >>$
(8) $\text{clé}(e) = c \Rightarrow \text{supprimer}(< e, l >, c) = l$
(9) $\text{clé}(e) < c \Rightarrow \text{supprimer}(< e, l >, c) =< e, \text{supprimer}(l, c) >$
(10) $\text{clé}(e) > c \Rightarrow \nexists l',\ l' = \text{supprimer}(< e, l >, c)$

Les opérations parcourent la liste tant que la clé de l'élément à rechercher, à supprimer ou à ajouter est supérieure à celle de l'élément courant.

Avec cette représentation, les trois opérations de base sont en $\mathcal{O}(n)$, avec une complexité moyenne égale à $\frac{1}{2}(n+1)$.

La position d'un nouvel élément à ajouter suit celle de l'élément qui lui est immédiatement inférieur. L'opération *ajouter* recherche cette position, puis modifie le chaînage, selon la même technique que pour la liste non ordonnée, afin d'insérer l'élément. Si cet élément est le plus petit, l'insertion est en tête de liste. Si la liste est représentée par un tableau, les opérations *supprimer* et *ajouter* sont plus coûteuses dans la mesure où elles doivent décaler une partie des éléments du tableau.

L'algorithme de recherche parcourt la liste de façon séquentielle jusqu'à ce que l'on ait trouvé un élément de clé supérieure ou égale à celle recherchée. En cas d'égalité, l'élément recherché est trouvé.

```
Algorithme rechercher(t, c)
    i ← 1
    trouvé ← faux
    tantque non trouvé et i≠longueur(t) faire
    {i≠longueur(t) et ∀k, 1≤k<i, clé(ième(t,k))<c}
        x ← ième(t,i)
        si clé(x)≥c alors
            trouvé ← vrai
          sinon i ← i+1
        finsi
    fintantque
    si clé(x)=c alors rendre x finsi
    {∀k, 1≤k≤longueur(t), clé(ième(t,k))≠c}
```

S'il y a équiprobabilité dans la recherche des éléments, le coût moyen d'une recherche positive et négative est $\frac{1}{2}(n+1)$. La complexité est $\mathcal{O}(n)$ comme pour la liste non ordonnée. La programmation en JAVA de cette méthode est donnée ci-dessous :

```
public Élément rechercher(Object clé)
throws CléNonTrouvéeException
{
  if (! comp.comparable(clé))
            throw new CléIncomparableException();
  Énumération énum=listeÉnumération();
  while (! énum.finÉnumération()) {
       Object x=énum.élémentSuivant();
       if (comp.supérieurOuÉgal(((Élément) x).clé(), clé))
           if (comp.égal(((Élément) x).clé(), clé))
                return (Élément) x;
           else
               // clé(élt courant)>clé
               throw new CléNonTrouvéeException();
  }
  // on est en fin de liste et
  // ∀k, 1≤k≤longueur(this), clé(ième(this,k))≠c
  throw new CléNonTrouvéeException();
}
```

La complexité des méthodes de recherche séquentielle dans des listes ordonnées ou non n'est pas très bonne puisqu'elle est de l'ordre de n. Toutefois, elles mettent en jeu des al-

gorithmes très simples, qui peuvent être raisonnablement utilisés pour des tables de moins d'une dizaine d'éléments.

20.3.3 Recherche dichotomique

Le principe de l'algorithme est de diviser l'espace de recherche de l'élément en deux espaces de même taille. L'élément recherché est dans l'un des deux espaces. La recherche se poursuit dans l'espace qui contient l'élément recherché selon la même méthode. Cette méthode de résolution par *partition* est une méthode classique qui consiste à diviser un problème en sous-problèmes de même nature mais de taille inférieure.

La recherche dichotomique nécessite une table ordonnée et, pour être efficace, un accès direct à chaque élément de la table. La représentation habituelle de la table est le tableau. Au début, l'espace de recherche est la liste entière, depuis un rang *gauche* égal à 1 jusqu'à un rang *droit* égal à la longueur de la table. Le rang du *milieu* $(gauche + droit)/2$ divise la table en deux. Si la clé recherchée est égale à celle de l'élément du milieu alors la recherche s'achève avec succès, sinon si elle lui est inférieure la recherche se poursuit dans l'espace de gauche, sinon elle lui est supérieure et la recherche à lieu dans l'espace de droite. La recherche échoue lorsque l'espace de recherche devient vide, c'est-à-dire lorsque les rangs *gauche* et *droit* se sont croisés. Une écriture évidente de cette méthode est :

```
Algorithme rechercher(t, c)
   gauche ← 1
   droit ← longueur(t)
   répéter
      {gauche≤droit et
       ∀k, 1≤k<gauche, clé(ième(t,k))<c et
       ∀k, droit<k≤longueur(t), clé(ième(t,k))>c}
      milieu ← (gauche+droit)/2
      x ← ième(t,milieu)
      si c=clé(x) alors  rendre x
      sinon
          si c<clé(x) alors droit ← milieu-1
            sinon {c>clé(x)} gauche ← milieu+1
          finsi
      finsi
   jusqu'à gauche>droit
   {∀k, 1≤k≤longueur(t), clé(ième(t,k))≠c}
```

Une autre version de cet algorithme est donné ci-dessous. Si l'élément est trouvé, les deux bornes *gauche* et *droit* sont modifiées de telle façon que la boucle s'achève et *milieu* indique alors le rang de l'élément recherché.

```
Algorithme rechercher(t, c)
   gauche ← 1
   droit ← longueur(t)
   {l'espace de recherche est au départ toute la table}
   répéter
      {gauche≤droit et
```

```
        ∀k, 1⩽k<gauche, clé(ième(t,k))<c et
        ∀k, droit<k⩽longueur(t), clé(ième(t,k))>c }
      milieu ← (gauche+droit)/2
      x ← ième(t,milieu)
      si c⩾clé(x) alors gauche ← milieu+1 finsi
      si c⩽clé(x) alors droit ← milieu-1 finsi
   jusqu'à gauche>droit
   {clé(ième(t,milieu))=c ou
    ∀k, 1⩽k⩽longueur(t), clé(ième(t,k))≠c}
   si clé(x)=c alors rendre x finsi
   {∀k, 1⩽k⩽longueur(t), clé(ième(t,k))≠c}
```

L'écriture de cet algorithme peut être amélioré en regroupant les deux tests d'égalité en un seul :

```
Algorithme rechercher(t, c)
   gauche ← 1
   droit ← longueur(t)
   répéter
      {gauche<droit et
       ∀k, 1⩽k<gauche, clé(ième(t,k))<c et
       ∀k, droit⩽k⩽longueur(t), clé(ième(t,k))⩾c}
      milieu ← (gauche+droit)/2
      x ← ième(t,milieu)
      si c⩽clé(x) alors droit ← milieu
      sinon {c>clé(x)} gauche ← milieu+1
   jusqu'à gauche⩾droit
   {clé(ième(t,gauche))=c ou
    ∀k, 1⩽k⩽longueur(t), clé(ième(t,k))≠c}
   x ← ième(tab,gauche)
   si clé(x)=c alors rendre x finsi
   {∀k, 1⩽k⩽longueur(t), clé(ième(t,k))≠c}
```

Vous remarquerez que cet algorithme ne s'arrête pas lorsque l'élément recherché est trouvé ! Au contraire, il se comporte de la même façon que s'il recherchait un élément n'appartenant pas à la table. Si l'on sait que la plupart des recherches sont négatives, cette écriture est très efficace dans la mesure où l'algorithme ne fait plus qu'une seule comparaison de clé à chaque itération. D'autre part, si un même élément apparaît plusieurs fois dans la table, cet algorithme permet de rechercher sa *première occurrence*, c'est-à-dire l'élément le plus à gauche dans la table.

Dans une table à n éléments, le nombre de comparaisons pour une recherche négative ou positive (dans le pire des cas) est égal à $\lfloor \log_2 n \rfloor + 1$. Cela correspond au parcours entier d'une branche d'un arbre binaire fortement équilibré.

Le nombre moyen de comparaisons est d'environ $\log_2 n - 1$ pour une recherche positive, et $\log_2 n$ pour une recherche négative. D'une façon générale, les algorithmes de recherche dichotomique ont une complexité égale à $\mathcal{O}(\log_2 n)$.

Les ajouts et les suppressions d'éléments en table par la méthode dichotomique ne sont pas efficaces dans la mesure où ils nécessitent des décalages traités de façon séquentielle. La méthode dichotomique est donc adaptée à des tables qui n'évoluent pas ou très peu, comme par exemple, la table des mots réservés d'un langage dans un compilateur. Lorsque les tables doivent évoluer, nous préférerons une représentation avec des arbres binaires.

20.4 REPRÉSENTATION PAR UN ARBRE ORDONNÉ

La table est représentée par un arbre binaire ordonné, c'est-à-dire un arbre binaire *étiqueté* dont les valeurs sont rangées suivant une relation d'ordre. Par la suite, nous considérerons la relation suivante entre les nœuds :

$$\forall a \in \mathcal{A}rbre_{bo}, \forall e \in \text{sag}(a), \forall e' \in \text{sad}(a)$$
$$\text{clé}(e) \leqslant \text{clé}(\text{valeur}(\text{racine}(a))) < \text{clé}(e')$$

Toutes les clés du sous-arbre gauche sont inférieures ou égales à la clé du nœud, elle-même strictement inférieure à toutes les clés du sous-arbre droit.

Dans le chapitre précédent, nous avons déjà évoqué les arbres binaires et les algorithmes associés. Nous avons vu que pour tout arbre binaire ayant n nœuds et une profondeur p, on a la double inégalité suivante :

$$\lfloor \log_2 n \rfloor \leqslant p \leqslant n - 1$$

La pire des recherches dans un arbre dégénéré est alors semblable à celle dans une liste linéaire ordonnée. Alors que pour un arbre parfaitement équilibré, elle sera égale à $\lfloor \log_2 n \rfloor + 1$. La complexité peut donc varier de $\mathcal{O}(n)$ à $\mathcal{O}(\log_2 n)$.

La profondeur moyenne d'un arbre est de l'ordre de $\sqrt{n}$. Le nombre de comparaisons moyen pour la recherche, l'ajout et la suppression d'un élément dans un arbre binaire de recherche quelconque est environ égal à $2 \log_2 n$.

Dans la programmation en JAVA des méthodes de la table, un arbre binaire ordonné sera représenté par une structure chaînée dont la définition a été donnée à la section 19.3.2.

```
public class ArbreOrdonnéChaîné implements Table {
    protected Class typeDesÉléments;
    protected ArbreBinaire r; // racine de l'arbre binaire ordonné
    protected Comparable comp;
    public ArbreOrdonnéChaîné(Comparable cmp) {
        typeDesÉléments=Élément.class;
        r=ArbreBinaireChaîné.arbreVide;
        comp=cmp;
    }
}
```

20.4.1 Recherche d'un élément

Nous noterons $a =< n,g,d >$ l'arbre a qui possède un nœud n, un sous-arbre gauche g et un sous-arbre droit d. À partir cette notation, les axiomes de la recherche s'expriment comme suit :

$\forall a \in Arbre_{bo}$, et $\forall c \in Clé$
(1) $\nexists e, e =$ rechercher(arbrevide,c)
(2) $c =$ clé(valeur(n)) $\Rightarrow$ rechercher($< n,g,d >$,c) = valeur(n)
(3) $c <$ clé(valeur(n)) $\Rightarrow$ rechercher($< n,g,d >$,c) = rechercher(g,c)
(4) $c >$ clé(valeur(n)) $\Rightarrow$ rechercher($< n,g,d >$,c) = rechercher(d,c)

La programmation en JAVA donnée ci-dessous correspond au modèle récursif des axiomes, et est semblable à celui de la recherche dichotomique. Si la clé de l'élément du nœud courant est la clé recherchée alors la recherche s'achève sur un succès, sinon elle se poursuit récursivement dans le sous-arbre gauche, si la clé recherchée est inférieure à celle du nœud courant, ou dans le sous-arbre droit dans le cas contraire. La recherche échoue lorsque un arbre vide est atteint.

```
public Élément rechercher(Object clé)
throws CléNonTrouvéeException
{
   if (! comp.comparable(clé))
       throw new CléIncomparableException();
   // rechercher depuis la racine de l'arbre
   return rechercher(r,clé);
}

private Élément rechercher(ArbreBinaire a, Object clé)
throws CléNonTrouvéeException
{
   if (a.estVide())
       throw new CléNonTrouvéeException();
   Object cléDuNoeud=((Élément) a.racine().valeur()).clé();
   if (comp.égal(cléDuNoeud,clé))
       // l'élément recherché est trouvé
       return (Élément) a.racine().valeur();
   // poursuivre la recherche
   return comp.supérieur(cléDuNoeud,clé) ?
       rechercher(a.sag(),clé) : rechercher(a.sad(),clé);
}
```

Cette programmation avec deux méthodes est nécessaire pour éviter le test de compatibilité de clé à chaque appel récursif.

20.4.2 Ajout d'un élément

Il existe plusieurs façons d'ajouter un élément dans une arbre binaire ordonné, mais toutes doivent maintenir la relation d'ordre. La plus simple consiste à placer l'élément à l'extrémité

d'une des branches de l'arbre. Ce nouvel élément est donc une feuille. Les axiomes suivants définissent l'opération *ajouter* en feuille.

$\forall a \in \mathcal{A}rbre_{bo}, et \forall e \in \mathcal{E}$

(5) ajouter(arbrevide,e) $=<$ e, arbrevide, arbrevide $>$

(6) clé(e) $\leqslant$ clé(valeur(n)) $\Rightarrow$ ajouter($< n,g,d >$,e) $=<$ n, ajouter(g,e),$d >$

(7) clé(e) $>$ clé(valeur(n) $\Rightarrow$ ajouter($< n,g,d >$,e) $=<$ n,g, ajouter(d,e) $>$

La programmation en JAVA de ces axiomes est donnée ci-dessous. Notez l'utilisation des méthodes `changerSag` et `changerSad` pour changer la valeur du sous-arbre gauche ou droit d'un arbre.

```
public void ajouter(Élément e) {
    if (! comp.comparable(e.clé))
        throw new CléIncomparableException();
    r=ajouter(r,e);
}
private ArbreBinaire ajouter(ArbreBinaire a, Élément e) {
   if (a.estVide()) a=new ArbreBinaireChaîné(e);
   else
     if (comp.supérieurOuÉgal(
              ((Élément) a.racine().valeur()).clé(),e.clé()))
         // clé du nœud courant ⩾ e.clé
         // ajouter dans le sous-arbre gauche
         a.changerSag(ajouter(a.sag(),e));
     else
         // clé du nœud courant < e.clé
         // ajouter dans le sous-arbre droit
         a.changerSad(ajouter(a.sad(),e));
   return a;
}
```

Malheureusement, l'ajout en feuille a l'inconvénient majeur de construire des arbres dont la forme dépend des séquences d'ajouts. Dans le pire des cas, si la suite d'éléments est croissante (ou décroissante), l'arbre engendré est dégénéré.

20.4.3 Suppression d'un élément

L'opération qui consiste à supprimer un élément de l'arbre est légèrement plus complexe. Si l'élément à retirer est un nœud qui possède au plus un sous-arbre, cela ne pose pas de difficulté. En revanche, s'il possède deux sous-arbres, il y a un problème : il faut lier ses deux sous-arbres à un seul point ! La solution pour contourner cette difficulté est de remplacer l'élément à enlever par le plus grand élément du sous-arbre gauche (ou le plus petit élément du sous-arbre droit) et de supprimer ce dernier. Dans les deux cas, la relation d'ordre est conservée.

$\forall a \in \mathcal{A}rbre_{bo}, \forall e \in \mathcal{E}$, et $\forall c \in \mathcal{C}lé$

(8) $\nexists a,\ a =$ rechercher(arbrevide,c)

(9) $c <$ clé(valeur(n)) $\Rightarrow$ supprimer($< n,g,d >$,c) $=< c,$ supprimer(g,c),$d >$
(10) $c >$ clé(valeur(n)) $\Rightarrow$ supprimer($< n,g,d >$,c) $=< n,g,$ supprimer(d,c) $>$
(11) $c =$ clé(valeur(n)) $\Rightarrow$ supprimer($< n,g,$ arbrevide $>$,c) $= g$
(12) $c =$ clé(valeur(n)) $\Rightarrow$ supprimer($< n,$ arbrevide ,$d >$,c) $= d$
(13) $g,d \neq$ arbrevide et $c =$ clé(valeur(n)) $\Rightarrow$
supprimer($< n,g,d >$,c) $=<$ max(g), supprimer(g, clé(max(g)),$d >$

avec la fonction *max* définie comme suit :

max : $\mathcal{A}rbre_{bo} \rightarrow \mathcal{E}$

(14) max($< n,g,$ arbrevide $>$) $=$ valeur(n)
(15) $d \neq$ arbrevide , max($< n,g,d >$) $= max(d)$

La programmation de l'opération *supprimer* suit la définition axiomatique précédente.

```
public void supprimer(Object clé) throws CléNonTrouvéeException
{
    if (! comp.comparable(clé))
        throw new CléIncomparableException();
    r=supprimer(r,clé);
}
private ArbreBinaire supprimer(ArbreBinaire a, Object clé)
{
    if (a.estVide()) // arbre vide ⇒ clé non trouvée
        throw new CléNonTrouvéeException();
    Object cléDuNoeud=((Élément) a.racine().valeur()).clé();
    if (comp.inférieur(clé,cléDuNoeud))
        a.changerSag(supprimer(a.sag(),clé));
    else
        if (comp.supérieur(clé,cléDuNoeud))
            a.changerSad(supprimer(a.sad(),clé));
        else // cléDuNoeud=clé ⇒ l'élément est trouvé
            if (a.sad().estVide()) // a=sag
               a=a.sag();
            else
                if (a.sag().estVide()) // a=sad
                    a=a.sad();
                else // a possède deux sous-arbres non vides
                    a.changerSag(suppmax(a.sag(),a.racine()));
    return a;
}
private ArbreBinaire suppmax(ArbreBinaire a, NoeudBinaire n)
{
    if (a.sad().estVide()) {
        a.changerSad(suppmax(a.sad(), n));
        return a;
    }
    else { // la racine de a est le max
```

```
            n.changerValeur(a.racine().valeur());
            return a.sag();
        }
}
```

Les opérations d'ajout et de suppression que vous venons de présenter ne donnent aucune garantie sur la forme des arbres qu'elles retournent, et peuvent en particulier, conduire à des performances en temps linéaires, semblables à celles des listes lorsque les arbres sont dégénérés. Pour garantir systématiquement une complexité logarithmique, il faut adapter les méthodes d'ajout et de suppression d'éléments afin qu'elles conservent l'arbre *équilibré*.

20.5 LES ARBRES AVL

Un arbre AVL[3] est un arbre binaire équilibré tel que pour n'importe quel de ses sous-arbres, appelons-le a, la différence de hauteur de ses sous-arbres gauche et droit n'excède pas un. Cette propriété est exprimée par l'inégalité suivante :

$$|\,\text{hauteur}(\text{sag}(a)) - \text{hauteur}(\text{sad}(a))| \leqslant 1.$$

La figure 20.1 montre un exemple d'arbre à sept nœuds qui possède la propriété AVL :

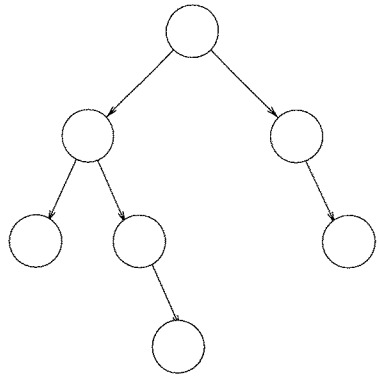


Figure 20.1 – *Un arbre AVL à sept nœuds .*

Pour un ensemble de n nœuds, la hauteur d'un arbre AVL est *toujours* $\mathcal{O}(\log_2 n)$. Plus précisément, ADELSON-VELSKII et LANDIS ont montré (cités par [Wir76]) que la hauteur h d'un arbre AVL qui possède n nœuds est liée par la relation :

$$\log_2(n+1) \leqslant h \leqslant 1{,}4404 \log_2(n+2) - 0{,}328.$$

3. Les arbres AVL doivent leur nom aux initiales de leurs auteurs, ADELSON-VELSKII et LANDIS, deux informaticiens russes qui les inventèrent en 1962.

20.5.1 Rotations

Les arbres AVL servent à représenter les arbres binaires ordonnés et garantissent une complexité de recherche de l'ordre de $\mathcal{O}(\log_2 n)$. En revanche, les opérations d'ajout et de suppression de la section 20.4 peuvent créer un déséquilibre qui remet en cause la propriété des arbres AVL. La différence de hauteur des sous-arbres qui violent la propriété est alors égale à 2. Pour conserver la propriété AVL, ces opérations doivent rééquilibrer l'arbre au moyen de deux types de *rotation*, les rotations *simples* et les rotations *doubles*. Ces rotations doivent bien sûr conserver la relation d'ordre sur les clés.

Considérons l'arbre (a) de la figure 20.2. Le nœud dont la clé a pour valeur 1 provoque un déséquilibre à gauche et réclame une restructuration de l'arbre, appelée *rotation simple à gauche*. Cette figure montre l'arbre (b) obtenu après cette rotation.

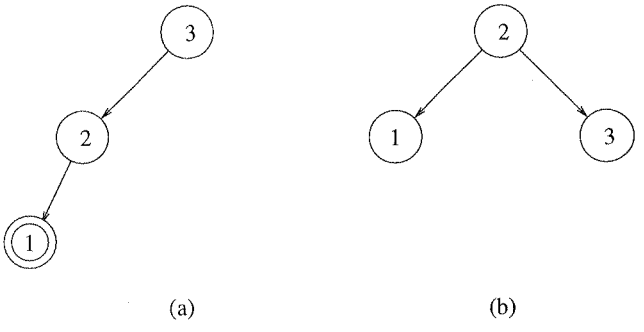


Figure 20.2 – *Une rotation simple à gauche.*

La figure 20.3 montre le cas général d'un déséquilibre dû au sous-arbre le plus à gauche (a), et la rotation simple à gauche qui permet de rééquilibrer l'arbre afin de retrouver la propriété AVL (b).

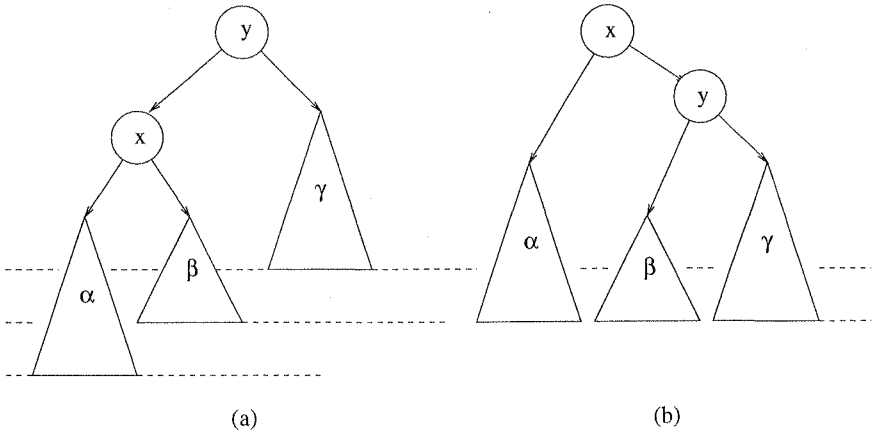


Figure 20.3 – *Rotation simple à gauche.*

De façon symétrique, un déséquilibre se produit lorsque le sous-arbre le plus à droite est trop grand. Le rééquilibrage de l'arbre est assuré par une rotation simple à droite, strictement symétrique à la précédente comme le montre la figure 20.4.

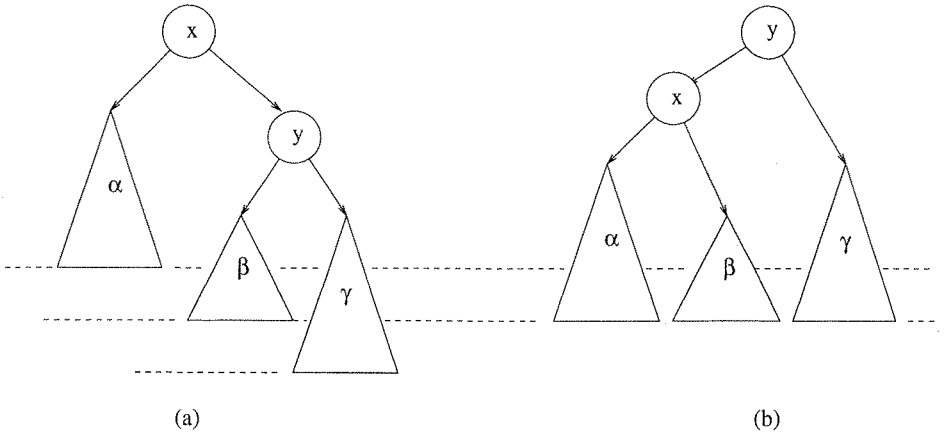


Figure 20.4 – *Rotation simple à droite.*

Les deux rotations simples précédentes ne peuvent résoudre toutes les formes de déséquilibre. Prenons le cas de l'arbre (a) de la figure 20.5. L'élément de clé 2 viole la propriété AVL et provoque un déséquilibre dans le sous-arbre droit du sous-arbre gauche de l'arbre. La réorganisation de l'arbre (b) est effectuée à l'aide d'une *rotation double à gauche*. Une rotation double à gauche consiste à appliquer successivement deux rotations simples : une rotation simple à droite du sous-arbre gauche, suivie d'une rotation simple à gauche de l'arbre.

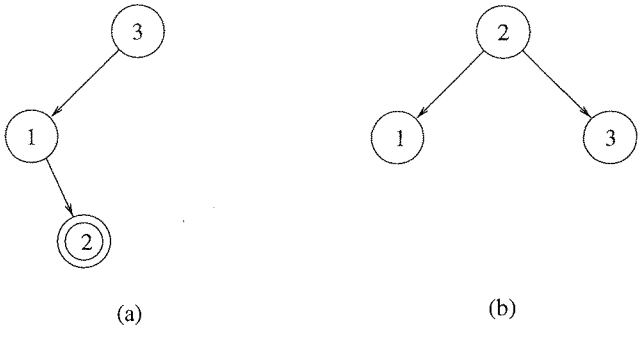


Figure 20.5 – *Une rotation double à gauche.*

La figure 20.6 montre la forme générale d'un arbre auquel il faut appliquer une rotation double à gauche pour le rééquilibrer.

Comme pour la rotation simple à gauche, il existe une rotation double à droite, symétrique de la gauche comme le montre la figure 20.7. Celle-ci consiste en une rotation simple à gauche du sous-arbre droit, suivie d'une rotation simple à droite de l'arbre.

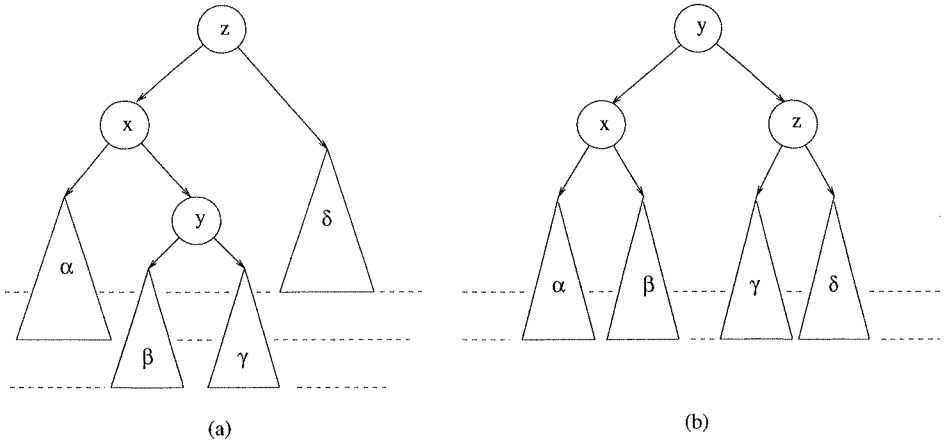


Figure 20.6 – *Rotation double à gauche.*

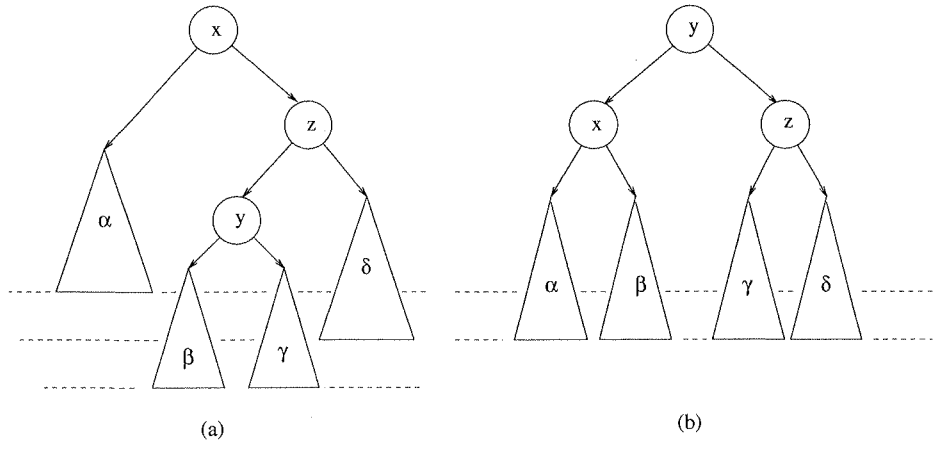


Figure 20.7 – *Rotation double à droite.*

Les opérations d'insertion et de suppression n'ont besoin que de ces quatre rotations pour maintenir la propriété AVL sur un arbre binaire ordonné.

Après l'ajout en feuille d'un nouvel élément, la vérification de la propriété AVL se fait en remontant la branche à partir de la nouvelle feuille insérée jusqu'à la racine de l'arbre. L'insertion provoque au maximum un déséquilibre et ne demande au plus qu'une rotation simple ou double pour rééquilibrer l'arbre. Le coût du rééquilibrage est $\mathcal{O}(1)$.

Après la suppression d'un élément, la vérification de la propriété AVL se fait, comme précédemment, en remontant vers la racine depuis le nœud supprimé. Mais, contrairement à l'insertion, une suppression peut provoquer (dans le pire des cas) une rotation de chaque nœud de la branche. Le coût du rééquilibrage est $\mathcal{O}(log_2 n)$.

D'un point de vue expérimental, la création d'arbres AVL ordonnés avec des clés données dans un ordre aléatoire, provoque environ une rotation (simple ou double en proportion égale) pour deux insertions, et une rotation (simple ou double, mais plus souvent simple que double) pour quatre suppressions.

20.5.2 Mise en œuvre

Les arbres AVL sont des arbres binaires sur lesquels les rotations décrites précédemment devront être possibles. Nous appellerons ces arbres binaires *arbres restructurables*. Pour les représenter, nous définissons l'interface `ArbreRestructurable` qui étend la classe `ArbreBinaire`. Elle propose à l'implémentation les méthodes qui assurent les quatre rotations. Cette interface est la suivante :

```
public interface ArbreRestructurable extends ArbreBinaire {
    public ArbreRestructurable rotationSimpleGauche();
    public ArbreRestructurable rotationSimpleDroite();
    public ArbreRestructurable rotationDoubleGauche();
    public ArbreRestructurable rotationDoubleDroite();
}
```

La mise en œuvre des arbres AVL s'effectue plus commodément à l'aide de structures chaînées. L'implémentation de l'interface `ArbreRestructurable` se fait par héritage de la classe `ArbreBinaireChaîné`. Les quatre méthodes de rotation modifient l'arbre courant selon les règles données dans la section précédente.

```
public class ArbreRestructurableChaîné extends ArbreBinaireChaîné
implements ArbreRestructurable
{
   public ArbreRestructurableChaîné(Object e)
   { super(e); }
   // Antécédent : l'arbre courant possède un sous-arbre gauche non vide
   // Rôle : effectue une rotation entre le nœud courant et
   // son sous-arbre gauche
   public ArbreRestructurable rotationSimpleGauche() {
       ArbreRestructurable a = (ArbreRestructurable) sag();
       changerSag(a.sad());
       a.changerSad(this);
       return a;
   }
   // Antécédent : l'arbre courant possède un sous-arbre droit non vide
   // Rôle : effectue une rotation entre le nœud courant et
   // son sous-arbre droit
   public ArbreRestructurable rotationSimpleDroite() {
       ArbreRestructurable a = (ArbreRestructurable) sad();
       changerSad(a.sag());
       a.changerSag(this);
       return a;
   }
   public ArbreRestructurable rotationDoubleGauche() {
       changerSag(((ArbreRestructurable) sag()).rotationSimpleDroite());
       return rotationSimpleGauche();
   }
```

```
    public ArbreRestructurable rotationDoubleDroite() {
        changerSad(((ArbreRestructurable) sad()).rotationSimpleGauche());
        return rotationSimpleDroite();
    }
} // fin classe ArbreRestructurableChaîné
```

La mise en œuvre d'une table à l'aide d'un arbre AVL ordonné est décrite par la classe `ArbreAVLChaîné`. Puisqu'un arbre AVL ordonné est un arbre ordonné particulier, cette classe hérite naturellement de la classe `ArbreOrdonnéChaîné`. Ceci nous permettra en particulier de ne pas avoir à récrire la méthode `rechercher`.

```
public class ArbreAVLChaîné extends ArbreOrdonnéChaîné
implements Table
{
    public ArbreAVLChaîné(Comparable cmp) { super(cmp); }
    ...
} // fin classe ArbreAVLChaîné
```

La propriété AVL requiert la connaissance de la hauteur des arbres manipulés. Pour des raisons d'efficacité, il n'est pas question de recalculer, par un parcours de l'arbre, cette hauteur chaque fois que cela est nécessaire. Au contraire, nous allons associer à chaque nœud un attribut qui mémorisera la hauteur de l'arbre. Cet attribut est simplement défini dans une classe `ÉlémentAVL`, héritière de la classe `Élément`.

```
public class ÉlémentAVL extends Élément {
    protected int hauteur = 0;
    public ÉlémentAVL(Élément e) {
        super(e.valeur(), e.clé());
    }
    public int hauteur() {
        return hauteur;
    }
    public void changerHauteur(int h) {
        hauteur=h;
    }
}
```

Les méthodes privées `rééquilibrerG` et `rééquilibrerD` sont utilisées par les méthodes `ajouter` et `supprimer`. Leur rôle est de calculer la nouvelle hauteur de l'arbre binaire a passé en paramètre et de vérifier s'il possède la propriété AVL. S'il y a un déséquilibre, elles procèdent à la rotation adéquate. Les nouvelles hauteurs des sous-arbres modifiés sont recalculées. Par convention, on considère que la hauteur d'un arbre vide est égale à -1. D'autre part, notez les conversions de type explicites de l'arbre binaire en arbre AVL.

```
private int hauteur(ArbreBinaire a) {
    return a.estVide() ? -1 :
           ((ÉlémentAVL) a.racine().valeur()).hauteur();
}
private void calculerHauteur(ArbreBinaire a) {
    ((ÉlémentAVL) a.racine().valeur()).changerHauteur(
                    1 + Math.max(hauteur(a.sag()), hauteur(a.sad())));
}
```

```
private ArbreBinaire rééquilibrerG(ArbreBinaire a) {
    if (hauteur(a.sag())-hauteur(a.sad())==2) {
        // l'arbre a n'est plus équilibré,
        // le sous-arbre gauche est trop grand
        a = hauteur(a.sag().sag())>=hauteur(a.sag().sad()) ?
            ((ArbreRestructurable) a).rotationSimpleGauche()
            :  // sinon rotation double à gauche
            ((ArbreRestructurable) a).rotationDoubleGauche();
        // calculer les hauteurs des nouveaux
        // sous-arbres gauche et droit de a
        calculerHauteur(a.sag()); calculerHauteur(a.sad());
    }
    // calculer la nouvelle hauteur de a
    calculerHauteur(a);
    return a;
}
private ArbreBinaire rééquilibrerD(ArbreBinaire a) {
    if (hauteur(a.sad())-hauteur(a.sag())==2) {
        // l'arbre a n'est plus équilibré,
        // le sous-arbre droit est trop grand
        a = hauteur(a.sad().sad())>=hauteur(a.sad().sag()) ?
            ((ArbreRestructurable) a).rotationSimpleDroite()
            : // sinon rotation double à droite
            ((ArbreRestructurable) a).rotationDoubleDroite();
        // calculer les hauteurs des nouveaux
        // sous-arbres gauche et droit de a
        calculerHauteur(a.sag()); calculerHauteur(a.sad());
    }
    // calculer la nouvelle hauteur de a
    calculerHauteur(a);
    return a;
}
```

Les méthodes `ajouter` et `supprimer` suivent les mêmes algorithmes récursifs que ceux donnés à la section 20.4 pour les arbres ordonnés simples, mais dans lesquels sont insérés les appels aux méthodes de vérification de la propriété AVL, `rééquilibrerG` et `rééquilibrerD`. Les vérifications sont faites après les appels récursifs, pour être exécutées en « remontant », du point d'insertion ou de suppression vers la racine.

```
private Object clé(ArbreBinaire a) {
    return ((Élément) a.racine().valeur()).clé();
}
public void ajouter(Élément e) {
    if (e.getClass()!=typeDesÉléments)
        throw new TypeIncompatibleException();
    if (! comp.comparable(e.clé()))
        throw new CléIncomparableException();
    r=ajouter(r,e);
}
private ArbreBinaire ajouter(ArbreBinaire a, Élément e) {
    if (a.estVide())
        return new ArbreRestructurableChaîné(new ÉlémentAVL(e));
```

```
        // a non vide
        if (comp.supérieurOuÉgal(clé(a),e.clé())) {
            // clé du nœud courant ⩾ e.clé
            // ajouter dans le sous-arbre gauche
            a.changerSag(ajouter(a.sag(),e));
            a=rééquilibrerG(a);
        }
        else {
            // clé du nœud courant < e.clé
            // ajouter dans le sous-arbre droit
            a.changerSad(ajouter(a.sad(),e));
            a=rééquilibrerD(a);
        }
        return a;
    }
    public void supprimer(Object clé) throws CléNonTrouvéeException {
        if (! comp.comparable(clé))
            throw new CléIncomparableException();
        r=supprimer(r, clé);
    }
    private ArbreBinaire suppmax(ArbreBinaire a, NoeudBinaire n) {
        if (!a.sad().estVide()) {
            a.changerSad(suppmax(a.sad(), n));
            return rééquilibrerG(a);
        }
        else { // la racine de a est le maximum
            n.changerValeur(a.racine().valeur);
            return a.sag();
        }
    }
    private ArbreBinaire supprimer(ArbreBinaire a, Object clé) {
        if (a.estVide()) // arbre vide ⇒ clé non trouvée
            throw new CléNonTrouvéeException();
        Object cléDuNoeud=clé(a);
        if (comp.inférieur(clé,cléDuNoeud)) {
            // clé du nœud courant ⩾ e.clé
            // supprimer dans le sous-arbre gauche
            a.changerSag(supprimer(a.sag(), clé));
            a=rééquilibrerD(a);
        }
        else
            if (comp.supérieur(clé,cléDuNoeud)) {
                // clé du nœud courant < e.clé
                // supprimer dans le sous-arbre droit
                a.changerSad(supprimer(a.sad(), clé));
                a=rééquilibrerG(a);
            }
            else // cléDuNoeud=clé ⇒ l'élément est trouvé
                if (a.sad().estVide()) // a=sag
                    a=a.sag();
```

```
            else
                if (a.sag().estVide()) // a=sad
                    a=a.sad();
                else { // a possède deux sous-arbres non vides
                    a.changerSag(suppmax(a.sag(), a.racine()));
                    a=rééquilibrerD(a);
                }
    return a;
}
```

20.6 ARBRES 2-3-4 ET BICOLORES

20.6.1 Les arbres 2-3-4

Les arbres que nous avons étudiés jusqu'à présent possèdent une seule clé (et sa valeur associée) par nœud. Mais en fait, rien n'interdit de mettre plusieurs clés dans un nœud, c'est justement cette possibilité qui garantira l'équilibre des arbres que nous allons présenter dans cette section.

On appelle « arbre 2-3-4 »[4] un arbre équilibré ordonné dont chaque nœud contient une, deux ou trois clés. Il est remarquable que toutes les branches, de la racine aux feuilles, de cet arbre possèdent la même longueur. Son nom vient du fait qu'un nœud à une clé possède deux sous-arbres, un nœud à deux clés possède trois sous-arbres, et un nœud à trois clés possède quatre sous-arbres. Aucun nœud interne ne possède de sous-arbres vides et nous appellerons ces nœuds, respectivement, nœud-2, nœud-3 et nœud-4.

Les clés dans un « arbre 2-3-4 » sont ordonnées de telle façon qu'un nœud-2 possède un sous-arbre pour les clés inférieures à sa clé, et un autre pour les clés supérieures. Un nœud-3 possèdent trois sous-arbres, un pour les clés inférieures à ses deux clés, un pour les clés comprises entre ses deux clés, et un troisième pour les clés supérieures à ses deux clés. Enfin, un nœud-4 possède quatre sous-arbres, un sous-arbre pour les clés inférieures à sa clé, un autre pour les clés supérieures, et deux autres pour les deux intervalles définis par ses trois clés. La figure 20.8 montre un exemple d'arbre 2-3-4, formé de neuf nœud-2, de quatre nœud-3 et de trois nœud-4, dont les clés sont ordonnées selon ces règles.

La forme parfaitement équilibrée des arbres 2-3-4 garantit une hauteur de l'arbre égale à $\lfloor \log_2 n \rfloor$, où n est le nombre de nœuds. Le coût d'une recherche est donc au pire égal à $3(\lfloor \log_2 n \rfloor + 1)$, si l'on procède à une recherche linéaire de l'élément dans chaque nœud. La complexité d'une recherche dans un arbre 2-3-4 est donc $\mathcal{O}(\log_2 n)$.

L'équilibre des arbres 2-3-4 est maintenu par les opérations d'ajout et de suppression de clés.

L'ajout d'un élément dans un arbre 2-3-4 se fait en feuille. Si cette feuille est un nœud-2 ou un nœud-3, l'élément est simplement inséré à sa place, ce qui transforme la feuille en un nœud-3 ou un nœud-4. En revanche, un problème se pose lorsque la feuille où ajouter le

4. Les arbres 2-3-4 sont des cas particuliers de B-arbres, arbres équilibrés inventés par R. BAYER et E. MCCREIGHT en 1972, pour permettre des recherches efficaces dans des tables placées en mémoire secondaire (*e.g.* disques).

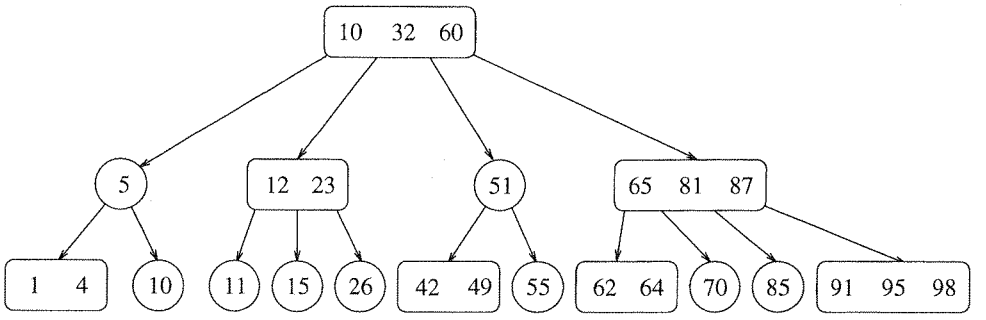


Figure 20.8 – *Un arbre 2-3-4.*

nouvel élément est un nœud-4. Il n'est en effet pas possible de placer un nouveau nœud-2 sous ce nœud-4 puisque cela remettrait en cause la forme équilibrée de l'arbre.

Une première solution, dite *ascendante*, consiste à scinder la feuille en deux nœud-2 dont les valeurs sont celles de l'élément de gauche et de droite du nœud-4, et d'insérer ensuite l'élément du milieu du nœud-4 dans son père. L'opération d'ajout peut alors placer le nouvel élément dans l'un des deux nouveaux nœud-2 créés en feuille. Le nœud père a été transformé en nœud-3 ou nœud-4, selon qu'il était au préalable un nœud-2 ou nœud-3. Si le père était lui-même un nœud-4, cette opération devra être renouvelée avec son propre père, et ainsi de suite jusqu'à la racine si nécessaire.

La seconde solution, dite *descendante*, scinde de façon similaire les nœud-4, mais cette fois-ci lors de la recherche de la feuille où insérer l'élément. Puisque les nœud-4 sont transformés en descendant la branche, le père de la feuille nœud-4 dans lequel on insère son élément central ne peut pas être un nœud-4. La figure 20.9 montre comment le nœud-4 (15,20,30) est scindé lorsque l'algorithme d'ajout le traverse. Lorsque la racine de l'arbre est un nœud-4, sa scission a pour effet de créer un nouveau nœud-2 et la hauteur de l'arbre est alors augmentée de un. Dans le pire des cas, si tous les nœuds de la branche sont des nœud-4, il faut procéder à $\lfloor \log_2 n \rfloor + 1$ scissions. Notez que la scission d'un nœud-4 est une opération relativement coûteuse, mais que de façon expérimentale, nous avons mesuré que les ajouts d'éléments dont les clés sont tirées de façon aléatoire provoquent en moyenne une scission pour deux ajouts.

Si on autorise l'ajout de plusieurs éléments de même clé, ceux-ci ne seront pas nécessairement placés côte à côte dans l'arbre. Une procédure de recherche qui renverrait l'ensemble des éléments de même clé devra alors poursuivre la recherche dans plusieurs sous-arbres à partir du nœud qui contient le premier élément avec la clé recherchée.

D'une façon générale, la suppression d'un élément consiste à le remplacer par l'élément de clé immédiatement inférieure[5]. Ce dernier se trouve nécessairement être le plus à droite dans la feuille la plus à droite en parcourant le sous-arbre des éléments de clés inférieures à celle de l'élément à supprimer. Si cette feuille est un nœud-3 ou un nœud-4, la suppression de l'élément le plus à droite ne pose aucune difficulté. En revanche, si la feuille est un nœud-2, il faut procéder soit à une *permutation*, soit à une *fusion*. Si son frère gauche est un nœud-3 ou un nœud-4, on fait une permutation d'éléments entre la feuille nœud-2, son père et son frère

5. On peut tout aussi bien choisir l'élément de clé immédiatement supérieure ou égale.

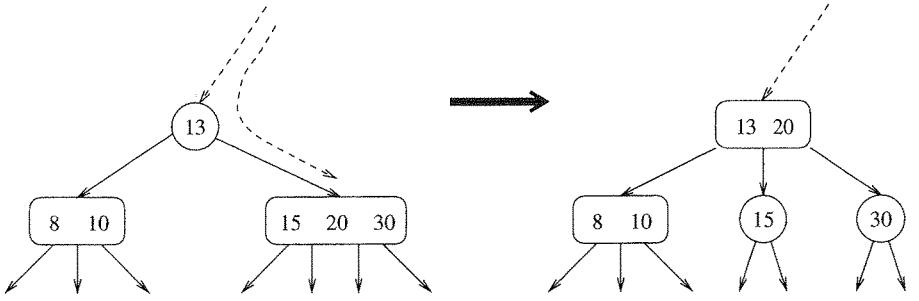


Figure 20.9 – *Éclatement d'un nœud-4.*

gauche, comme le montre par exemple la figure 20.10 dans le cas de la suppression de la clé 45.

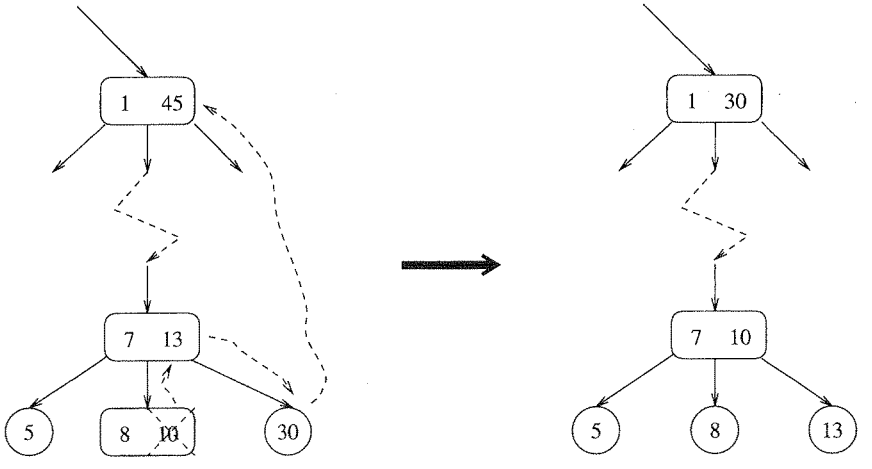


Figure 20.10 – *Suppression de la clé 45 provoquant une permutation.*

Si le frère gauche est un nœud-2, la permutation précédente n'est plus possible, et on remplace l'élément de la feuille nœud-2 par l'élément le plus à droite de son père, puis on procède à la fusion de la feuille nœud-2 et de son frère gauche. Le nombre d'éléments du nœud père est diminué de un. La figure 20.11 montre un exemple de fusion. Notez que si le père est lui même un nœud-2, il faut recommencer cette opération de fusion au niveau du père. De proche en proche, elle peut se propager jusqu'à la racine de l'arbre 2-3-4, et dans ce cas la hauteur de l'arbre est diminuée de un. Dans le pire des cas, le nombre de fusions est égal à la hauteur de l'arbre 2-3-4 plus un.

La propagation de la fusion souffre une exception dans le cas où le nœud à fusionner est un nœud-4. Par exemple dans l'arbre de la figure 20.12, la suppression de la clé 8 provoque la fusion des clés 5 et 10, et se propage sur le nœud père. La clé 13 remplace la clé 8, et doit être fusionnée avec son voisin (*i.e.* 20 21 35). Mais, ce nœud est un nœud-4 et il n'est pas possible, par définition, de créer un nœud-5 (*i.e.* 13 20 31 45). La solution consiste alors à

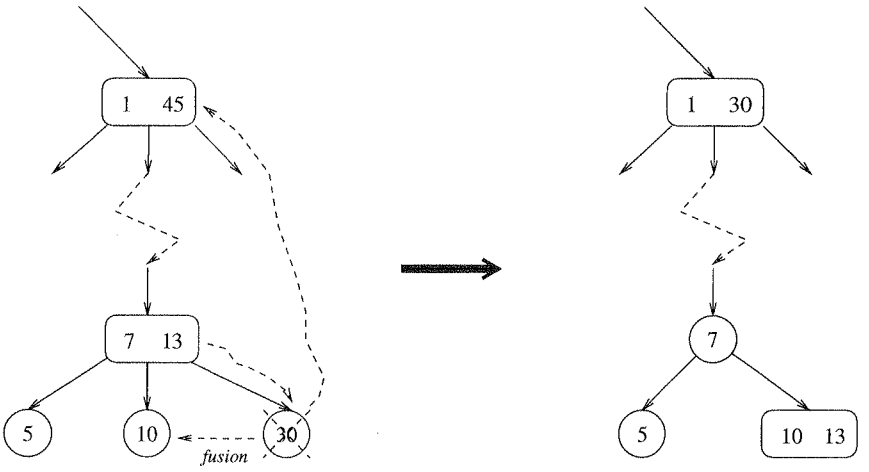


Figure 20.11 – *Suppression de la clé 45 provoquant une fusion.*

transférer le premier élément du nœud-3 dans son père. Si le père était lui-même un nœud-2, on poursuivrait par une fusion avec son voisin (dans l'exemple, 13 avec 31 et 45).

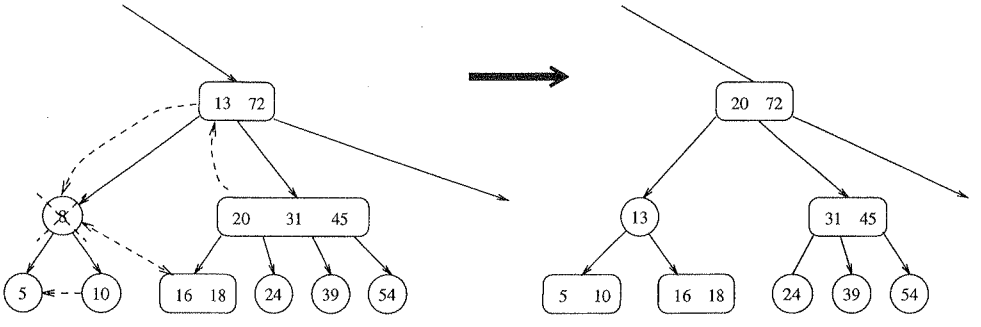


Figure 20.12 – *Suppression de la clé 8.*

Les résultats expérimentaux de suppressions aléatoires de clés montrent environ un transfert pour quatre suppressions et une fusion pour six suppressions.

20.6.2 Mise en œuvre en Java

Il est possible de représenter les arbres 2-3-4 par des arbres généraux tels que nous les avons définis au chapitre 19. La mise en œuvre de l'interface `Table` à l'aide d'un arbre 2-3-4 est décrite par la classe `Arbre234`.

```
public class Arbre234 implements Table {
    protected Arbre r; // racine de l'arbre 2-3-4
    protected Class typeDesÉléments;
    protected Comparable comp;
```

```
    public Arbre234(Comparable cmp) {
        typeDesÉléments=Élément.class;
        r=null;
        comp=cmp;
    }
    ...
}
```

Les éléments et leurs clés, contenus dans chaque nœud de l'arbre, sont rangés dans une liste ordonnée mise en œuvre à l'aide d'un tableau à trois composants. On les représente par une classe locale `Valeur234` qui hérite de `ListeOrdonnéeTableau`. On munit cette classe d'une méthode particulière de recherche linéaire qui renvoie le rang dans la liste de l'élément recherché si celui-ci est présent, ou celui de l'élément qui lui est immédiatement supérieur dans le cas contraire.

```
class Valeur234 extends ListeOrdonnéeTableau {
    Valeur234(Élément e, Comparable cmp)  throws ClassNotFoundException
    {
        super(3,cmp);
        super.ajouter(1,e);
    }
    int rechercher(Comparable cmp, Object clé) {
        int i=1;
        while (i<=longueur() &&
                        cmp.supérieur(clé, ((Élément) ième(i)).clé()))
            i++;
        return i;
    }
}
```

La méthode `rechercher` dans un arbre 2-3-4 est donnée ci-dessous. Dans chaque nœud, on effectue la recherche linéaire de la clé. Si cette recherche échoue, `r` donne le rang du ième sous-arbre dans lequel doit se poursuivre récursivement la recherche ; mais si le nœud courant est une feuille, la recherche dans l'arbre s'achève et échoue.

```
public Élément rechercher(Object clé) throws CléNonTrouvéeException {
    if (! comp.comparable(clé))
        throw new CléIncomparableException();
    if (estVide()) // la table est vide
        throw new CléNonTrouvéeException();
    return rechercher(r, clé);
}
private Élément rechercher(Arbre a, Object clé)
throws CléNonTrouvéeException
{
    // rechercher dans la clé dans le nœud courant
    Valeur234 n=(Valeur234) a.valeur();
    int r=n.rechercher(comp, clé);
    // r>n.longueur() ou clé≤n.ième(r).clé()
    if (r<=n.longueur() && comp.égal(clé,((Élément) n.ième(r)).clé()))
        // clé trouvée
        return (Élément) n.ième(r);
```

```
    // sinon clé<n.ième(r).clé()
    if (!a.estUneFeuille())
       // a n'est pas une feuille ⇒ poursuivre la recherche
        return rechercher(a.forêt().ièmeArbre(r), clé);
    // sinon clé non trouvée
    throw new CléNonTrouvéeException();
}
```

Les méthodes `ajouter` et `supprimer` sont plus délicates à programmer, et plus particulièrement la seconde. Elles constituent d'ailleurs un excellent exercice de programmation laissé au lecteur.

20.6.3 Les arbres bicolores

La mise en œuvre précédente d'un arbre 2-3-4 par un arbre général rend les algorithmes d'ajout et de suppression complexes et finalement moins efficaces que ceux des arbres AVL, et peut-être même que ceux des arbres binaires simples. Au chapitre 19, nous avons vu que tout arbre général possédait une représentation binaire. Les arbres bicolores, appelés également arbres *rouge-noir*, sont des arbres binaires ordonnés qui offrent une représentation particulièrement efficace des arbres 2-3-4.

Pour bien comprendre les opérations d'ajout et de suppression décrites dans cette section, il est essentiel d'avoir clairement à l'esprit la correspondance entre les nœuds des arbres bicolores avec ceux des arbres 2-3-4. Les équivalents binaires de chaque type de nœud d'un arbre 2-3-4 sont donnés par la figure 20.13. Les frères dans un nœud-3 ou un nœud-4 sont coloriés en rouge (liens gras et cercles doubles dans la figure), alors que l'aîné est colorié en noir (liens maigres et cercles simples). La figure 20.14 montre les représentations 2-3-4 et bicolore d'un arbre dans lequel on a inséré les clés de 1 à 8 en ordre croissant.

À partir de ces règles de correspondance entre nœuds, il est très facile de déduire que la racine d'un arbre bicolore est noire, qu'un nœud rouge est nécessairement le fils d'un nœud noir, et que toutes les branches de l'arbre possèdent le même nombre de nœuds noirs. Ce dernier point est dû au fait que toutes les branches d'un arbre 2-3-4 ont la même longueur, et qu'il ne peut y avoir dans un arbre bicolore qu'un seul nœud noir par nœud 2-3-4. Enfin, on en déduit que la hauteur d'un arbre bicolore à n éléments est au plus égale au double de l'arbre 2-3-4 correspondant, c'est-à-dire qu'elle est $\mathcal{O}(\log_2 n)$.

La méthode de recherche dans un arbre bicolore est identique à celle dans un arbre binaire standard, et la forme équilibrée de l'arbre garantit une complexité de recherche de l'ordre de $\mathcal{O}(\log_2 n)$ pour un arbre à n éléments.

L'ajout d'un élément dans un arbre bicolore se fait en feuille et pose le même problème que dans un arbre 2-3-4. Il s'agit de scinder les équivalents binaires des nœud-4, soit lors de la recherche de la feuille où insérer l'élément (algorithme descendant), soit après l'insertion dans une feuille nœud-4 (algorithme ascendant). Pour un arbre bicolore, l'opération de scission consiste simplement à colorier en noir les deux frères, et en rouge l'aîné puisque ce dernier est inséré dans son père. La figure 20.15 montre cette opération de changement de couleur.

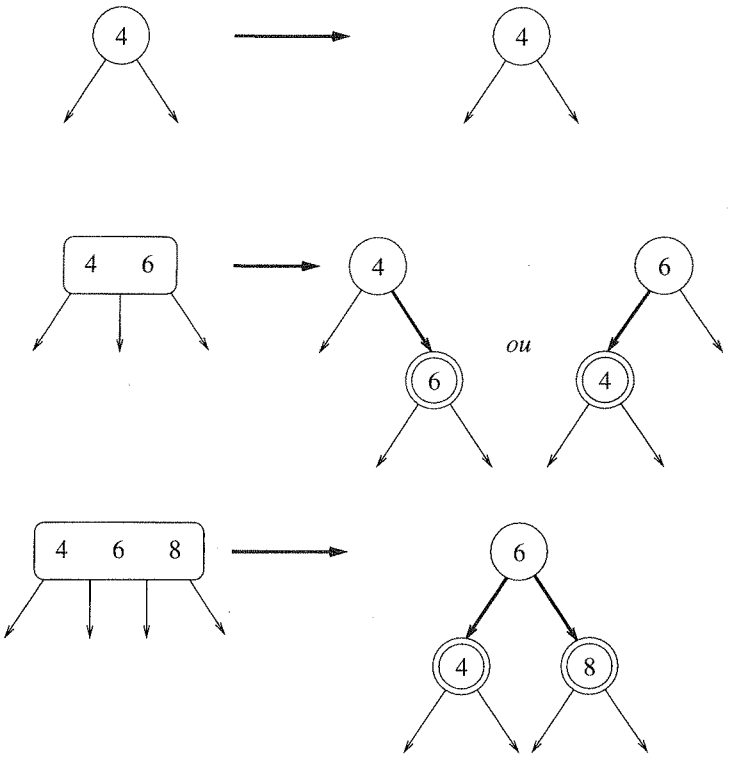


Figure 20.13 – *Correspondances entre nœuds 2-3-4 et nœuds bicolores.*

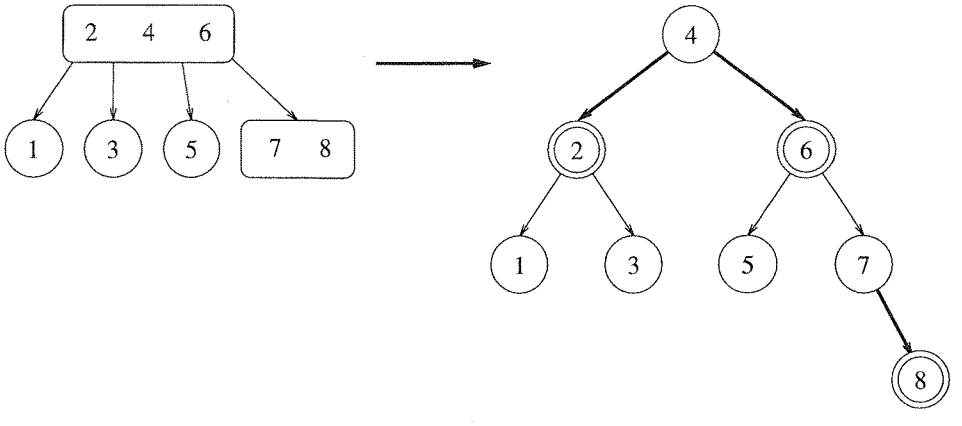


Figure 20.14 – *Le même arbre sous forme 2-3-4 et bicolore.*

Notez que si l'aîné est la racine de l'arbre, ce nœud devra être colorié à nouveau en noir, puisque le changement de couleur correspond à la scission d'un nœud-4 à la racine d'un arbre 2-3-4. Souvenez-vous que c'est l'unique cas où la hauteur d'un arbre 2-3-4 est augmentée de un.

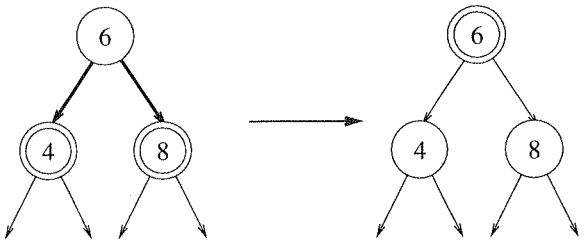


Figure 20.15 – *Changement de couleur d'un nœud-4.*

Nous avons vu qu'un arbre bicolore ne peut posséder deux nœuds rouges consécutifs. L'opération de changement de couleur précédente peut violer cette règle si le père du nœud qui a été colorié en rouge est lui-même un nœud rouge. On élimine ce problème par les rotations simples ou doubles identiques à celles présentées dans la section 20.5.1. La figure 20.16 montre les quatre situations possibles et les rotations à faire. Notez que le grand-père est par définition nécessairement un nœud noir. Après chaque rotation, le père est colorié en noir, et le grand-père en rouge.

Pour qu'on puisse supprimer un élément, celui-ci doit être présent dans l'arbre bicolore. Comme pour un simple arbre binaire ordonné, il ne sera possible de supprimer qu'un nœud ayant au plus un sous-arbre. Si le nœud qui le contient possède deux sous-arbres, on remplacera l'élément à supprimer par l'élément de clé immédiatement inférieure ou égale[6] situé le plus à droite dans le sous-arbre gauche. C'est ce nœud avec au plus un sous-arbre qui sera supprimé. Sa suppression consiste simplement à lier son père avec son unique fils gauche.

Si le nœud supprimé est rouge, ou si son fils unique est rouge (le nœud supprimé était donc noir), on colorie en noir le fils et la suppression est terminée. Notez que cela correspond à la suppression d'un élément dans un nœud-3 ou un nœud-4. Dans le cas contraire, le nœud supprimé et son fils unique sont noirs. Cela correspond à la suppression d'un nœud-2 dans un arbre 2-3-4, que nous avions traitée par une permutation ou une fusion selon la nature de son frère. Pour un arbre bicolore, nous envisagerons trois cas de figure.

Premier cas, le frère du nœud supprimé est noir et possède un fils rouge. Le frère est un équivalent nœud-3 ou nœud-4, et la suppression était assurée par une permutation. Pour un arbre bicolore, il faut produire une rotation simple ou double, du fils rouge, du frère et du père du nœud supprimé. Après la rotation, le nouveau père prend la couleur de l'ancien père, ses fils gauche et droit sont coloriés en noir, tout comme le fils unique. La figure 20.17 montre un exemple de permutation, et sa correspondance avec un arbre 2-3-4 ; la suppression de l'élément 30 provoque une rotation double à gauche.

Deuxième cas, le frère du nœud supprimé est noir et ses deux fils sont noirs, c'est-à-dire que le frère est l'équivalent d'un nœud-2. On colorie en noir le fils unique et en rouge le frère. Si le père du nœud supprimé est rouge, il suffit de colorier ce père en noir, pour obtenir l'effet d'une fusion dans l'arbre équivalent 2-3-4. La figure 20.18 met en évidence ce cas lorsqu'on supprime l'élément 13. En revanche, si le père est noir et qu'il n'est pas la racine de l'arbre, on considère à nouveau les trois cas de figure à son niveau. Ceci consiste à faire

6. Ou de clé immédiatement supérieure ou égale.

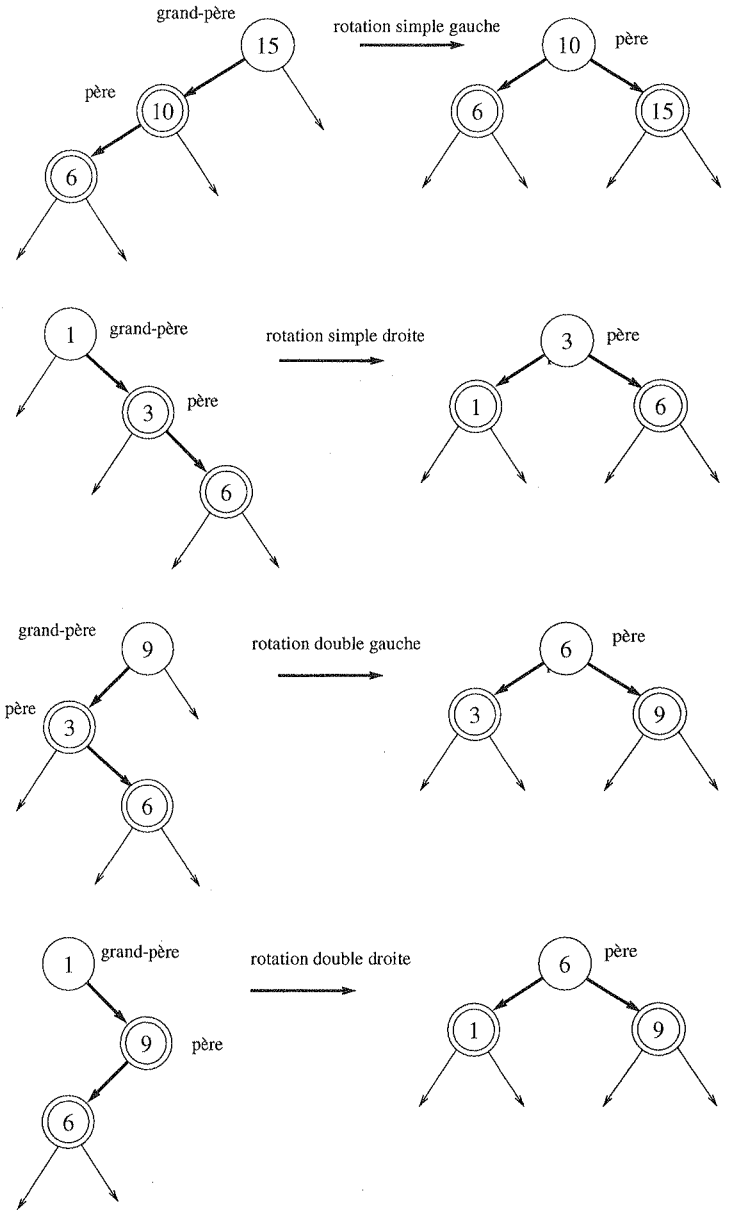


Figure 20.16 – *Suppression de 2 nœuds rouges consécutifs.*

une propagation au niveau du parent. Le nombre de propagations possibles est la hauteur de l'arbre, $\mathcal{O}(\log_2 n)$.

Troisième cas, le frère du nœud supprimé est rouge. Il s'agit de changer sa couleur en noir pour être en mesure d'appliquer ensuite l'un des deux cas précédents. Pour cela, on

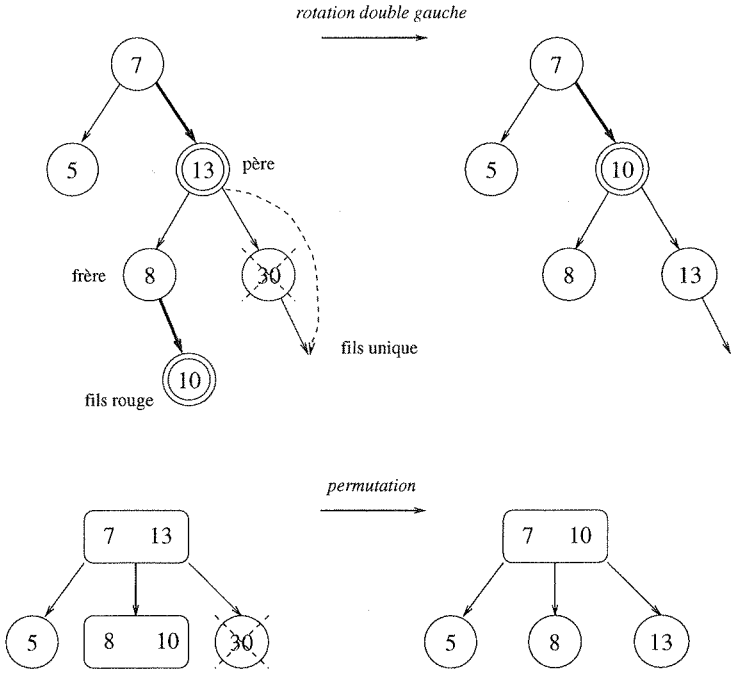


Figure 20.17 – *Suppression de 30 dans l'arbre bicolore, et sa correspondance 2-3-4.*

transforme l'arbre à l'aide d'une rotation simple. La rotation simple fait intervenir le père du nœud supprimé, le frère, et son fils gauche (si le frère est un fils gauche) ou son fils droit dans le cas contraire. Après la rotation, le frère devient le nouveau père, il est colorié en noir, et l'ancien père est colorié en rouge. Notez qu'après cette transformation, l'application du deuxième cas n'entraînera pas de propagation puisque le père est rouge. Dans la figure 20.19, le frère droit de l'élément 5 supprimé est rouge. La rotation simple à droite réorganise l'arbre de telle façon que le nouveau frère 13 est un nœud noir. Le deuxième cas peut maintenant être appliqué.

La complexité de la suppression est $\mathcal{O}(\log_2 n)$. Pour chaque suppression, il y a $\mathcal{O}(\log_2 n)$ propagations, et au plus une réorganisation et une permutation, soit une rotation simple, plus une rotation simple ou double.

Les mesures expérimentales d'ajout de clés dans un ordre aléatoire donnent environ deux rotations, une simple et une double, pour cinq insertions (ce qui est un peu mieux que pour un arbre AVL). Pour les suppressions, le nombre de rotations est semblable à celui des arbres AVL, c'est-à-dire une rotation pour quatre suppressions.

20.6.4 Mise en œuvre en Java

Comme les arbres AVL, les arbres bicolores sont représentés par des arbres restructurables chaînés sur lesquels les rotations, simples et doubles, sont possibles. L'implantation d'une

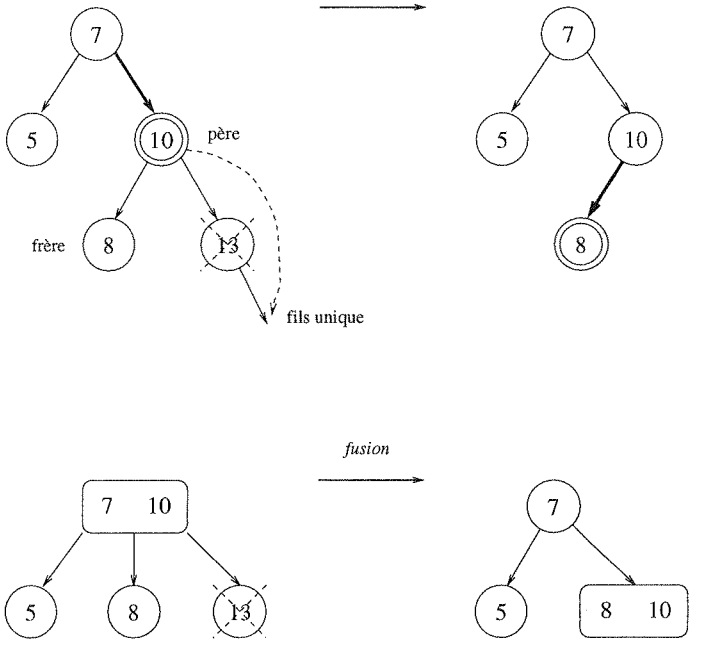


Figure 20.18 – *Suppression de 13 dans l'arbre bicolore, et sa correspondance 2-3-4.*

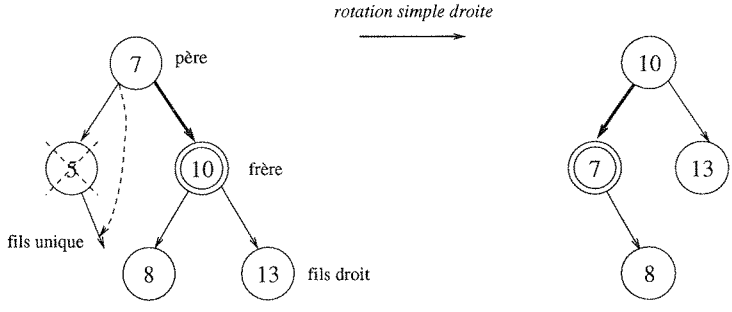


Figure 20.19 – *Suppression de 5 dans l'arbre bicolore.*

table à l'aide d'un arbre bicolore est décrite par la classe `ArbreBicoloreChaîné` qui hérite de la classe `ArbreOrdonnéChaîné` et redéfinit ses deux méthodes `ajouter` et `supprimer`, la méthode `rechercher` étant obtenue par héritage.

```
public class ArbreBicoloreChaîné extends ArbreOrdonnéChaîné
implements Table
{
    public ArbreBicoloreChaîné(Comparable cmp)
    { super(cmp); }
    ...
}
```

La couleur des nœuds est conservée dans un attribut associé à chacun des éléments des nœuds de l'arbre. Comme pour les arbres AVL, on définit une classe héritière de la classe `Élément`, que nous appellerons `ÉlémentBicolore`, dans laquelle on déclare un attribut `couleur` et les méthodes qui permettent d'accéder à sa valeur ou de la changer. Notez que, par convention, la couleur d'un arbre vide est noire.

```
public class ÉlémentBicolore extends Élément {
    public static final int NOIR=0;
    public static final int ROUGE=1;
    protected int couleur;
    public ÉlémentBicolore(Élément e, int c) {
        super(e.valeur(), e.clé());
        couleur=c;
    }
    public ÉlémentBicolore(Élément e) {
        super(e.valeur(), e.clé());
        couleur=ROUGE;
    }
    public int couleur() { return couleur; }
    public void changerCouleur(int c) {
        couleur=c;
    }
}
```

La méthode `ajouter`, donnée ci-dessous, parcourt de façon itérative la branche à l'extrémité de laquelle le nouvel élément sera ajouté. En plus du nœud courant, elle mémorise les nœuds père, grand-père et arrière-grand-père à l'aide de trois attributs privés. Lors de la recherche de la feuille où à lieu l'insertion, chaque équivalent binaire d'un nœud-4 (racine des sous-arbres gauche et droit rouges) est scindé (algorithme descendant) grâce à l'appel de la méthode privée `scinderNoeud`. Un dernier appel à cette méthode est nécessaire après l'ajout en feuille de l'élément.

```
private ArbreBinaire père, grandPère, arrièreGrandPère;
public void ajouter(Élément e) {
    if (e.getClass()!=typeDesÉléments)
        throw new TypeIncompatibleException();
    if (! comp.comparable(e.clé()))
        throw new CléIncomparableException();
    if (r.estVide())
        r=new ArbreRestructurableChaîné(
                        new ÉlémentBicolore(e, ÉlémentBicolore.NOIR));
    else {
        // l'arbre n'est pas vide
        ArbreBinaire courant;
        courant=père=grandPère=arrièreGrandPère=r;
        do {
            if (estRouge(courant.sag()) && estRouge(courant.sad()))
                // équivalent nœud-4 à scinder
                scinderNoeud(courant);
            arrièreGrandPère=grandPère;
            grandPère=père;
```

```
            père=courant;
            courant =
                comp.supérieurOuÉgal(clé(courant), e.clé()) ?
                // clé du nœud courant ⩾ e.clé
                // ajouter dans le sous-arbre gauche
                courant.sag() :
                // clé du nœud courant < e.clé
                // ajouter dans le sous-arbre droit
                courant.sad();
        } while (!courant.estVide());
        // père désigne la feuille où attacher le nouveau nœud
        if (comp.supérieurOuÉgal(clé(père), e.clé())) {
            père.changerSag(new ArbreRestructurableChaîné(
                                                    new ÉlémentBicolore(e)));
            courant=père.sag();
        }
        else {
            père.changerSad(new ArbreRestructurableChaîné(
                                                    new ÉlémentBicolore(e)));
            courant=père.sad();
        }
        scinderNoeud(courant);
    }
}
```

La méthode privée `scinderNoeud` commence par changer les couleurs du nœud courant, ainsi que ceux de ses sous-arbres gauche et droit selon la règle donnée plus haut. Si son père est rouge, il y a donc deux nœuds rouges consécutifs, et une rotation simple ou double est alors nécessaire pour rendre l'arbre à nouveau bicolore. C'est la méthode `restructurer` qui choisit la rotation à appliquer. Après la rotation, le père et le grand-père changent à leur tour de couleur.

```
private void scinderNoeud(ArbreBinaire a) {
    enRouge(a); enNoir(a.sag()); enNoir(a.sad());
    if (a == r) enNoir(r);
    else
        // y a-t-il deux nœuds rouges consécutifs?
        // note: si père=r, pas de rotation car la racine est noire
        if (estRouge(père)) {
            ArbreBinaire np=restructurer(a, père,
                                        grandPère, arrièreGrandPère);
            // changer les couleurs
            enNoir(np); enRouge(np.sag()); enRouge(np.sad());
        }
}
```

Le type de rotation à effectuer est fonction de la structure de l'arbre. La méthode `restructurer` suivante applique les rotations simples ou doubles comme l'indique la figure 20.16 à la page 268.

```
private ArbreBinaire restructurer(ArbreBinaire a, ArbreBinaire p,
                                  ArbreBinaire gp, ArbreBinaire agp)
```

```
{
    if (p.sag() == a)
        p = gp.sag() == p ?
              ((ArbreRestructurable) gp).rotationSimpleGauche()
            : ((ArbreRestructurable) gp).rotationDoubleDroite();
    else // (p.sad() == a)
        p = gp.sad() == p  ?
              ((ArbreRestructurable) gp).rotationSimpleDroite()
            : ((ArbreRestructurable) gp).rotationDoubleGauche();

    // accrocher l'arbre rééquilibré à l'arrière-grand-père s'il existe
    // sinon, le père est la nouvelle racine de l'arbre
    if (gp == r) r=p;
    else
        if (agp.sad() == gp) agp.changerSad(p);
        else agp.changerSag(p);
    return p;
}
```

La méthode de supprimer est donnée ci-dessous. La propagation nécessite la connaissance des ascendants du nœud supprimé. Ceux-ci sont mémorisés dans une pile lors du parcours de la branche qui conduit au nœud à supprimer. La fonction parent retourne le père du nœud courant, situé en sommet de pile.

```
public void supprimer(Object clé) throws CléNonTrouvéeException {
    if (! comp.comparable(clé))
        throw new CléIncomparableException();
    if (estVide()) // arbre vide ⇒ clé non trouvée
        throw new CléNonTrouvéeException();
    // l'arbre n'est pas vide
    Pile lesPères=new PileChaînée(ArbreRestructurableChaîné.class);
    ArbreBinaire courant=r;  boolean trouvée=false;
    do {
        if (comp.égal(clé(courant), clé)) trouvée=true;
        else {
            lesPères.empiler(courant);
            courant = (comp.supérieurOuÉgal(clé(courant), clé)) ?
                    courant.sag()  // clé du nœud courant ⩾ clé
                  : courant.sad(); // clé du nœud courant < clé
        }
    } while (!trouvée && !courant.estVide());
    // courant désigne le nœud contenant la clé à supprimer
    // OU clé non trouvée
    if (!trouvée) throw new CléNonTrouvéeException();
    // rechercher le maximum du sous-arbre gauche de courant
    ArbreBinaire noeudSupp =courant, suivant=noeudSupp.sag();
    while (!suivant.estVide()) {
        lesPères.empiler(noeudSupp);
        noeudSupp=suivant; suivant=suivant.sad();
    }
    // noeudSupp désigne le nœud à supprimer contenant la clé
```

```
// immédiatement inférieure ou égale à celle de courant
if (r == noeudSupp) {
    // on supprime la racine, son sous-arbre gauche
    // est nécessairement vide
    r=noeudSupp.sad(); enNoir(r);
    return;
}
int couleurEltSupp=couleur(noeudSupp),
    ancienneCouleurCourant=couleur(courant);
// mettre la valeur de noeudSupp dans courant
if (courant!=noeudSupp) {
    courant.racine().changerValeur(noeudSupp.racine().valeur());
    changerCouleur(courant, ancienneCouleurCourant);
}
// prendre le fils unique du nœud supprimé,
// son père et son grand-père
ArbreBinaire filsNoeudSupp = (courant == noeudSupp) ?
                             noeudSupp.sad() : noeudSupp.sag();
père=parent(lesPères);
grandPère=parent(lesPères);
// suppression de noeudSupp ⇒
// à quel sous-arbre du père accrocher son fils?
if (père.sad() == noeudSupp) père.changerSad(filsNoeudSupp);
  else père.changerSag(filsNoeudSupp);
if (estRouge(filsNoeudSupp)||couleurEltSupp==ÉlémentBicolore.ROUGE)
     enNoir(filsNoeudSupp);
else // le nœud supprimé est noir et son fils aussi
    do {
        ArbreBinaire frère=quelFrère(filsNoeudSupp, père);
        if (estRouge(frère)) {
            // le frère est rouge ⇒ réorganisation
            ArbreBinaire filsFrèreRouge=
                    frère == père.sad() ? frère.sad() : frère.sag();
            grandPère =
               restructurer(filsFrèreRouge, frère, père, grandPère);
            enNoir(grandPère); enRouge(père);
        } else { // le frère est noir
            if (estRouge(frère.sag()) || estRouge(frère.sad())) {>
               // un de ses fils est rouge ⇒ permutation
               ArbreBinaire filsRouge =
               estRouge(frère.sag()) ? frère.sag() : frère.sad(),
                a=restructurer(filsRouge, frère, père, grandPère);
               changerCouleur(a, couleur(père));
               enNoir(a.sag()); enNoir(a.sad());
               enNoir(filsNoeudSupp);
               return;
            }
            // ces deux fils sont noirs
            enNoir(filsNoeudSupp); enRouge(frère);
            if (estRouge(père)) { // fusion
```

```
                    enNoir(père);
                    return;
                } else { // propagation si le père n'est pas la racine
                    if (père == r) return;
                    filsNoeudSupp=père;
                    père=grandPère;
                    grandPère=parent(lesPères);
                }
            }
        } while (true);
} // fin supprimer
```

20.7 TABLES D'ADRESSAGE DISPERSÉ

L'idée des tables d'adressage dispersé[7] est d'accéder, en une seule comparaison, à un élément de la table grâce à une fonction calculée à partir de sa clé. Cette fonction est utilisée par les opérations d'ajout, de recherche et de suppression.

De façon formelle, il s'agit de définir une fonction h, appelée *fonction d'adressage*, telle que :

$$h \quad : \quad Clé \quad \rightarrow \mathcal{P}lace$$

où $\mathcal{P}lace$ est l'ensemble des positions possibles pour un élément dans la table. Le plus souvent, les tables d'adressage dispersé sont représentées par des tableaux, et le rôle de la fonction h est de retourner une valeur prise dans le type des indices du tableau. La convention habituelle est de choisir les indices sur l'intervalle $[0,m-1]$, où m est le nombre de composants du tableau. Une fonction d'adressage « idéale » retourne une place différente pour chacune des clés des éléments.

Prenons une table qui possède onze places, dans laquelle on désire insérer sept éléments dont les clés alphabétiques sont les suivantes : *Louise*, *Germaine*, *Paul*, *Maud*, *Pierre*, *Jacques* et *Monique*. L'évaluation d'une fonction d'adressage idéale pour ces clés retourne, par exemple, les sept indices suivants 6, 2, 8, 3, 1, 7 et 10. La figure 20.20 page 276 montre la table avec les éléments à leur place (pour simplifier, seule la clé est affichée).

Notez que les éléments dans les tables d'adressage ne sont pas ordonnés et, contrairement aux représentations de table vues précédemment, le coût de la recherche dans une table d'adressage dispersé est *constant* et ne dépend pas du nombre d'éléments dans la table. Pour une fonction d'adressage idéale, ce coût est égal au coût de calcul de la fonction h, auquel s'ajoute celui de la vérification de la présence ou l'absence de l'élément recherché à la place calculée.

Ces méthodes sont très souvent utilisées par les compilateurs pour implanter les tables de symboles, ou par les correcteurs orthographiques des logiciels de traitement de texte. Elles

7. Appelées en anglais, *hash tables*, car il s'agit de hacher ou de découper la clé pour pour n'en conserver qu'une partie utile à la recherche, l'ajout ou la suppression d'un élément dans la table. On trouve souvent dans la littérature française la traduction littérale « tables de hachage » pour les désigner. Nous préférons employer le terme « table d'adressage dispersé » qui nous semble plus évocateur du principe général de la méthode.

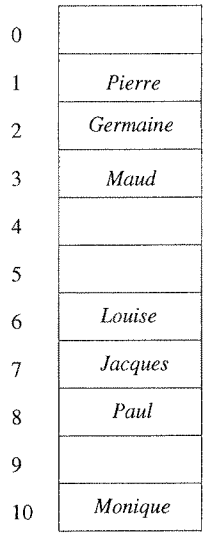


Figure 20.20 – *Une table d'adressage dispersé.*

sont très performantes à condition, toutefois, de bien choisir la fonction d'adressage, la taille de la table et la façon de traiter les collisions.

20.7.1 Le problème des collisions

Bien souvent la fonction d'adressage est une surjection, c'est-à-dire que pour deux clés différentes, la fonction h renvoie une même place. On dit qu'il y a une *collision* lorsque

$$\exists\, c_1, c_2 \in Clé,\ c_1 \neq c_2 \text{ et } h(c_1) = h(c_2)$$

Les collisions sont en général inévitables[8], et nous verrons à la section 20.7.3 la façon de les résoudre. [FGS90] montre en particulier, sous certaines hypothèses probabilistiques, que la fonction h disperse uniformément sur l'ensemble des places de la table, et que la probabilité que la fonction h retourne la même place pour cinq clés différentes avoisine zéro. Il en résulte que d'une façon générale la recherche d'un élément dans une table d'adressage dispersé réclame au plus cinq accès quelle que soit la taille de la table. Des résultats statistiques établis sur divers ensembles de clés et plusieurs fonctions d'adressage semblent confirmer de façon expérimentale cette probabilité théorique [ASU89].

Le choix de la fonction h est donc primordial. Il doit être tel qu'il minimise le nombre de collisions. Ce choix peut être difficile à faire, surtout si l'on n'a pas de connaissance *a priori* sur les valeurs des clés. S'il est mauvais, il peut rendre catastrophique une méthode efficace.

8. Le célèbre paradoxe du jour anniversaire affirme que si plus de vingt-trois personnes sont réunies dans une même salle, il y a près d'une chance sur deux que deux d'entre elles soient nées le même jour.

20.7.2 Choix de la fonction d'adressage

Une bonne fonction d'adressage doit à la fois répartir les clés le plus uniformément sur l'ensemble des places de la table et être simple et rapide à calculer.

Toute fonction d'adressage doit d'abord passer à une représentation entière de la clé (sauf si la clé est déjà sous cette forme) et finir par rendre un indice calculé modulo m pour revenir sur l'intervalle $[0,m-1]$. En général, la clé est une chaîne de caractères et il est important d'en extraire seulement les caractères significatifs. En particulier, il est inutile de faire intervenir les caractères blancs lorsqu'ils terminent les chaînes. La représentation entière de la chaîne est obtenue avec des sommes arithmétiques ou, si l'opérateur est disponible dans le langage, des unions exclusives des caractères qui la composent. Voici quelques méthodes couramment utilisées :

- prendre quatre caractères au centre de la chaîne ;
- prendre les trois premiers et les trois derniers caractères de la chaîne ;
- additionner les entiers obtenus en regroupant les caractères par blocs de quatre caractères ;
- calculer des suites de la forme h_0, $h_i = \alpha h_{i-1} + c_i$.

Pour cette dernière façon de précéder, la suite est très simple à calculer et à programmer. D'ailleurs, c'est avec une fonction d'adressage de ce type que la table de la figure 20.20 a été produite. Nous en donnons la programmation en JAVA.

```
int fctAdressage(String s) {
    int h=0;
    for (int i=0; i<s.length(); i++)
            h = 32*h+s.charAt(i);
    return Math.abs(h)%m;
}
```

JAVA ne vérifie pas les dépassements de capacité des opérations arithmétiques. La valeur de la variable `h` à la fin de boucle peut donc être négative, voilà pourquoi il faut prendre sa valeur absolue. Pour les langages qui font la vérification, le calcul dans la boucle devra être fait modulo m, ce qui rend de fait cette fonction plus coûteuse à calculer. Notez qu'en JAVA, on peut remplacer avantageusement le produit par 32 par un décalage vers la gauche de cinq bits :

```
h = (h<<5) + s.charAt(i);
```

Il n'existe pas de fonction d'adressage universelle. En fait, elle dépend de l'ensemble des clés utilisé. Une fonction efficace pour des noms ou des prénoms ne le sera pas nécessairement pour les mots réservés d'un langage de programmation, ou pour des codes de sécurité sociale. Si l'ensemble des clés est connu, il est possible de chercher une fonction sans collision. Il existe d'ailleurs des générateurs de fonctions d'adressage qui, à partir d'un ensemble de clés connu, fournissent des fonctions sans (autant que faire se peut) collision [Sch90]. En revanche, si la nature des clés n'est pas connue, on choisira une fonction simple à calculer et dont on essaiera de prévoir le comportement à partir d'un échantillon de clés.

Le choix de la valeur m est également à prendre en considération. La méthode d'adressage dispersé est très efficace, mais ce gain de temps se fait au prix d'une certaine perte de

place mémoire par rapport aux méthodes qui ne requièrent que l'espace mémoire strictement nécessaire pour mémoriser les éléments. En général, la taille de la table est fixée à l'avance, et il faudra réserver un nombre de places supérieur au nombre d'éléments prévus. Si les éléments occupent un espace mémoire important, il ne faudra pas les conserver tels quels dans la table (en JAVA, par exemple, les éléments de la table seront des références).

Le taux de remplissage de la table, c'est-à-dire le rapport n/m, avec n le nombre effectif d'éléments présents dans la table, a une influence sur la qualité de la fonction d'adressage. Plus ce taux sera faible, plus le risque de collisions sera réduit, mais encore une fois au prix d'une perte de place mémoire.

Enfin, et ce n'est pas de moindre importance, pour éviter une accumulation de collisions pour les clés dont la représentation entière serait un multiple de m, la taille de la table doit être un nombre *premier*.

20.7.3 Résolution des collisions

Il n'est malheureusement pas toujours possible d'éviter les collisions, et l'opération *ajouter(t,e)* devra quand même insérer dans la table l'élément e, même si sa clé entre en collision avec celle d'un autre élément déjà présent dans la table. Il existe deux grandes techniques pour résoudre les collisions. La première utilise des fonctions secondaires, la seconde chaîne les collisions dans ou hors de la table.

➤ Utilisation de fonctions secondaires (adressage ouvert)

Cette méthode consiste à appliquer, en cas de collision, une seconde fonction d'adressage pour obtenir une nouvelle place *dans* la table. Si cette nouvelle place provoque à nouveau une collision, on applique une troisième fonction d'adressage, et ainsi de suite jusqu'à ce qu'il n'y ait plus de collision ou que le nombre de fonctions secondaires ait été épuisé. Plus formellement, cette méthode dispose d'un ensemble de fonctions h_i tel que :

$$h_i(c) = (h(c) + f(i)) \mod m, \text{ avec } f(0) = 0$$

La fonction f est la méthode de résolution des collisions. La première fonction h_0 appliquée à la clé est donc h, on l'appelle la fonction d'adressage *primaire*. Les autres fonctions h_i sont appelées *secondaires*. Pour une clé donnée, les fonctions secondaires fournissent une suite particulière de places, parmi toutes les suites possibles, c'est-à-dire $m!$.

Comme pour la fonction h, le choix de la fonction f est délicat. Le coût d'évaluation des fonctions secondaires doit rester faible. D'autre part, elles doivent disperser les éléments le plus uniformément possible sur toutes les places disponibles de la table. Les suites de places retournées par les fonctions h_i pour deux clés différentes doivent être si possible différentes. Ce qui distinguera les différentes fonctions f que nous allons présenter, c'est bien sûr leur pouvoir de dispersion des éléments dans la table.

Une première méthode, appelée *linéaire*, définit $f(i) = i$, c'est-à-dire que les fonctions secondaires sont telles que :

$$h_i(c) = (h(c) + i) \mod m$$

Cette méthode est particulièrement simple, puisqu'elle consiste à consulter $h(c) + 1$, $h(c) + 2$, $h(c) + 3$, ..., c'est-à-dire les places de distance i par rapport à $h(c)$. L'algorithme échoue et s'arrête lorsque $i = m$ puisque toutes les places de la table auront été consultées en vain. Cette méthode n'est toutefois pas très bonne car les séquences de places calculées pour deux clés différentes sont identiques, et elle provoque des accumulations d'éléments autour des clés qui ont été placées du premier coup, sans collision. [Knu73] donne une approximation du taux de collision pour cette méthode. Pour une recherche positive, ce taux est environ égal à $(1 - d/2)/(1 - d)$, où d est le taux de remplissage de la table.

Une variante de cette première méthode, appelée *quadratique*, doit son nom au fait que la fonction f a la forme d'une équation du second degré. Les fonctions secondaires sont telles que :

$$h_i(c) \quad = \quad (h(c) + a\,i^2 + b\,i) \mod m$$

Cette méthode évite le problème d'accumulation de la méthode linéaire, mais les séquences de places produites pour deux clés distinctes en collision restent identiques. La qualité de cette méthode dépend du choix de a et de b. Le meilleur choix pour éviter de chevaucher, autant que faire se peut, les suites de place retournées par d'autres clés, est de prendre a et b premier avec m. Notez qu'il sera toujours possible d'ajouter un élément si la table est au plus à moitié remplie. La plupart du temps, on choisit $a = 1$ et $b = 0$, car les fonctions secondaires sont alors très simples à calculer avec une relation de récurrence qui évite l'élévation au carré. Nous donnons la programmation en JAVA de la méthode rechercher.

```
// Recherche dans une table d'adressage dispersé
// résolution des collisions par fonctions secondaires
// méthode quadratique: i²
public Élément rechercher(Object clé) {
    if (! comp.comparable(clé))
            throw new CléIncomparableException();
    int d=1,
        h=fctPrimaire(clé);
    do {
        if (table[h]==null)
                throw new CléNonTrouvéeException();
        // la place h est occupée
        if (comp.égal(table[h].clé(), clé)) return table[h];
        // collision ⇒ appeler une fonction secondaire
        if (d>=m) // plus de fonctions secondaires
            throw new CléNonTrouvéeException();
        // passer à la fonction secondaire suivante
        h=(h+d)%m;
        d+=2;
    } while (true);
}
```

Notez que l'algorithme s'arrête lorsque $i^2 \geqslant m$. Dans l'hypothèse où m est premier, alors seulement $m/2$ places auront été testées. Toutefois, ceci est assez rare en pratique et n'arrive que lorsque le taux de remplissage de la table est élevé.

Une troisième méthode, appelée *adressage double*, consiste à disposer d'une seconde fonction d'adressage h'. La fonction f est telle que $f(i) = i\,h'(c)$, c'est-à-dire que les fonctions secondaires sont de la forme :

$$h_i(c) \quad = \quad (h(c) + i\,h'(c)) \mod m$$

Cette méthode évite les accumulations et, contrairement aux deux techniques précédentes, produit pour deux clés différentes en collision des suites de places distinctes. Pour un adressage le plus uniforme possible sur toute la table, $h'(c)$ doit être différent de 0 et premier avec m pour tout c. KNUTH [Knu73] suggère de choisir $h'(c) = 1 + (c \mod (m-2))$, avec m et $m-2$ premiers. Il indique également que le taux de collision pour une recherche positive, si les deux fonctions sont indépendantes et uniformes, est environ $-d^{-1}\log(1-d)$.

Chaînage externe des collisions

Avec cette méthode, et contrairement à la technique précédente, on ne cherche pas dans la table une nouvelle place pour les éléments dont les clés entrent en collision. Ceux-ci sont rangés dans des structures dynamiques en *dehors* de la table. Ces structures peuvent être des listes ou des arbres. Le coût de cette méthode est celui de la fonction d'adressage primaire, plus celui de la recherche, de l'ajout ou de la suppression de l'élément dans la structure de données choisie. En général, le nombre de collisions étant faible (pas plus de cinq pour une fonction primaire uniforme), une simple liste linéaire est tout à fait acceptable, et conserve à la méthode d'adressage dispersé toute son efficacité. Notez que pour une place donnée, le nombre moyen d'éléments chaînés est n/m.

Cette méthode a l'avantage d'être très simple à mettre en œuvre, et permet une gestion plus efficace de la mémoire. En effet, si la place mémoire est limitée[9], il est possible de réduire la taille de la table sans que cela pénalise trop le taux de collision.

Néanmoins, si pour une raison ou une autre, la fonction d'adressage provoquait un nombre de collisions anormal sur une même place dans la table, la méthode d'adressage dispersé perdrait tout son intérêt. Plus encore qu'avec l'utilisation de fonctions secondaires, il est important d'évaluer sur un échantillon de clés la dispersion de la fonction primaire utilisée.

Dans la méthode `ajouter` donnée ci-dessous, le chaînage externe est fait à l'aide d'une liste non ordonnée.

```
public void ajouter(Élément e) {
    if (!comp.comparable(e.clé()))
        throw new CléIncomparableException();
    int h=fctPrimaire(e.clé());
    if (table[h]==null) {
        // la place est libre ⇒ créer une nouvelle liste
        try {
            table[h]=new ListeNonOrdonnéeChaînée(comp);
        }
        catch (ClassNotFoundException excep) {}
    }
```

9. La taille des mémoires des ordinateurs actuels ne cessent d'augmenter, mais la taille des données manipulées aussi!

```
        // ajouter e dans la liste table[h]
        table[h].ajouter(e);
    }
```

Chaînage interne des collisions

Le chaînage des collisions se fait dans la table elle-même. On peut définir une zone d'adressage primaire et une zone secondaire pour le chaînage des collisions. Le choix de la taille des deux zones est la principale difficulté de cette méthode. Une autre solution est de n'avoir qu'une zone unique, mais alors, comme pour l'adressage ouvert, il peut y avoir création de collisions secondaires qui n'ont pas lieu d'être.

La figure 20.21 page 281 montre ces deux formes d'organisation de la table : (a) deux zones, (b) une seule zone. Une fonction d'adressage primaire a calculé les indices 1 et 3 pour les clés *Pierre* et *Maud*. Les clés *Louise* et *Jacques* entrent toutes les deux en collision avec *Maud*.

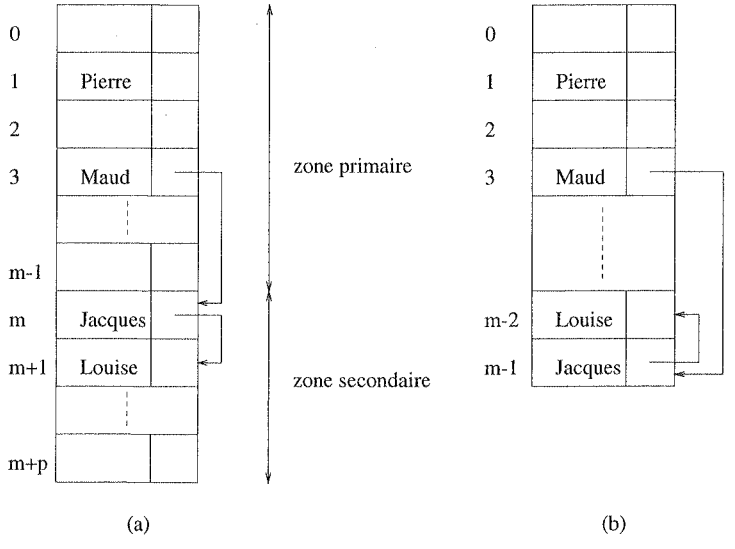


Figure 20.21 – *Chaînage interne.*

Notez qu'avec cette méthode, il est nécessaire de maintenir une liste des places libres à utiliser pour le chaînage des collisions. Dans la figure 20.21 (a), la zone secondaire comporte $p + 1$ places libres. Elle commence à la place m et progresse vers la place $m + p$. Dans la figure 20.21 (b), les éléments en collision sont chaînés à partir de la place $m - 1$ vers le début de la table. Par simplification, la liste des places libres n'a pas été représentée dans la figure.

Comparaisons des méthodes

Lorsque le taux de remplissage est relativement faible, le nombre de collisions primaires est très faible et toutes ces méthodes de résolution des collisions sont équivalentes. Mais lorsque ce taux est important et que le nombre de collisions augmente, le chaînage dans des

structures externes est préférable. L'adressage ouvert présente plusieurs inconvénients. D'une part, il peut attribuer une place à un élément, après la résolution de sa collision, qui peut très bien être celle d'un prochain élément calculée par la fonction primaire, et qui aurait donc été libre sans la collision précédente. D'autre part, les opérations de suppression d'éléments sont difficiles à mettre en œuvre. La suppression d'un élément qui provoque une collision nécessite le souvenir de la collision ou le déplacement des éléments qui sont en collision avec l'élément à supprimer. Au contraire, dans le cas d'un chaînage externe, l'opération de suppression est celle de l'élément dans la structure externe.

La méthode de chaînage externe possède également l'avantage de permettre la conservation d'un nombre d'éléments supérieur à m. En revanche, elle nécessite plus de place mémoire, ne serait-ce que celle utilisée pour le chaînage.

20.8 EXERCICES

Exercice 20.1. Rédigez les algorithmes qui mettent en œuvre les deux techniques de recherche auto-adaptative (donnée à la page 244) dans une liste linéaire non ordonnée.

Exercice 20.2. On décide de ranger les mots réservés d'un langage de programmation dans une table tmr. Ces mots sont rangés par ordre de longueurs croissantes et par ordre alphabétique pour les mots d'une même longueur. L'accès à cette table se fait par une autre table, appelée tdl, telle que $\forall i \in [1,longueur(tdl)]$ tous les mots de longueur i sont sur l'intervalle

$$[\text{ième}(tmr, \text{ième}(tdl,i)),\ \text{ième}(tmr, \text{ième}(tdl,i+1)-1)]$$

De plus, si $\text{ième}(tdl,i) = \text{ième}(tdl,i+1)$ alors le nombre de mots de longueur i est égal à 0.

Donnez l'algorithme d'une recherche d'un mot réservé. Quelle en ait sa complexité ? Puis, écrivez un algorithme qui construit la table des longueurs à partir de la table des mots réservés.

Exercice 20.3. Donnez une implantation de l'interface `Table` avec une liste non ordonnée représentée par un tableau, puis par une structure chaînée.

Exercice 20.4. Refaites l'exercice précédent avec une liste ordonnée.

Exercice 20.5. Une recherche dichotomique peut être utilisée pour rechercher la place d'un élément à insérer dans une table ordonnée. Modifiez les algorithmes de recherche dichotomique, afin qu'il retourne le rang de l'élément à insérer.

Exercice 20.6. La recherche dichotomique coupe en deux parties égales l'espace de recherche sans se préoccuper de la nature de la clé recherchée. Lorsque vous recherchez un mot dans un dictionnaire, vous l'ouvrez vers le début, le milieu ou la fin selon que le mot recherché commence par une lettre proche du début, du milieu ou de la fin de l'alphabet. La méthode de recherche par *interpolation* met en œuvre cette technique et améliore la méthode de recherche dichotomique. L'idée est d'estimer l'endroit où la clé pourrait se trouver sur la connaissance des valeurs disponibles. La figure 20.22 montre comment calculer le rang

estimé *inter* entre les rangs *gauche* et *droit* d'une liste l. Notez que l'élément recherché doit nécessairement appartenir à l'intervalle $[\text{ième}(l,1)), \text{ième}(l, \text{longueur}(l))]$.

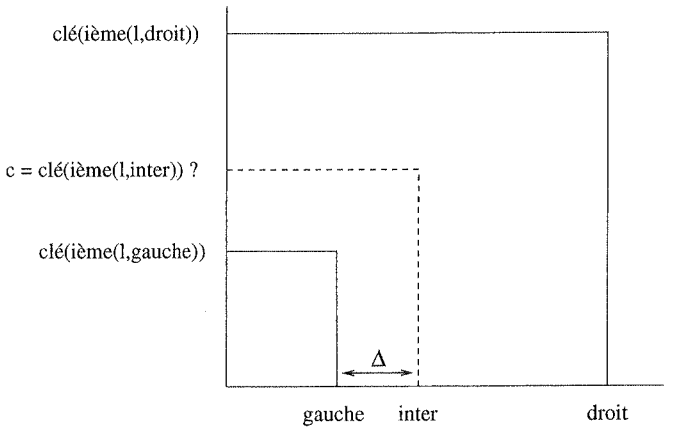


Figure 20.22 – *Le rang* inter *est égal à* gauche $+\Delta$.

Écrivez un algorithme de recherche par interpolation. Comparez cette méthode avec la recherche dichotomique.

Exercice 20.7. Écrivez de façon itérative l'algorithme de recherche dans un arbre binaire ordonné.

Exercice 20.8. L'opération de *coupure* divise un arbre binaire ordonné en deux arbres binaires ordonnés distincts par rapport à une clé c. Le premier arbre est tel que toutes les clés de ses nœuds sont inférieures ou égales à la clé c, le second arbre est tel que les clés de ses nœuds sont strictement supérieures à c. La figure 20.23 donne un exemple de coupure.

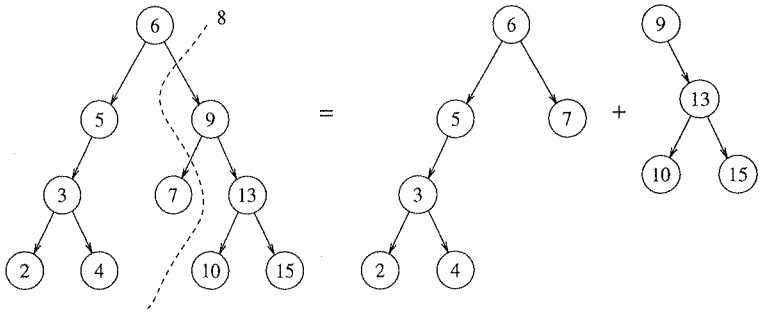


Figure 20.23 – *Coupure d'un arbre par rapport à la clé 8.*

Donnez la signature de cette opération. Écrivez les axiomes qui donnent formellement la sémantique de cette fonction, puis programmez-la en JAVA.

Exercice 20.9. L'opération *ajouter* dans un arbre binaire ordonné (donnée page 249) ajoute un élément en feuille. Utilisez l'opération *coupure* de l'exercice précédent pour définir une

opération d'ajout à la racine. Vous écrirez les axiomes de cette opération, et vous la programmerez en JAVA. Quelle est sa complexité dans le pire des cas ?

Exercice 20.10. On définit un arbre quaternaire ordonné, le type abstrait qui permet de ranger des valeurs appartenant à un espace à deux dimensions où chaque dimension est munie d'une relation d'ordre. Chaque nœud est le père de quatre sous-arbres qui représentent une partition de l'espace à deux dimensions, auquel appartient la valeur du nœud, en quatre quadrants. Si nous appelons les deux dimensions respectivement *latitude* et *longitude*, nous pourrons appeler les quatre quadrants, respectivement, sud-ouest (*SO*), nord-ouest (*NO*), sud-est (*SE*) et nord-est (*NE*).

Ainsi, étant donnée une valeur particulière qui occupe un nœud de l'arbre quaternaire, son sous-arbre sud-est ne comprend que des valeurs dont la latitude est inférieure ou égale à la sienne et la longitude strictement supérieure à la sienne. Remarquez la dissymétrie des relations, nécessaire pour que l'on puisse placer sans hésitation des points distincts mais qui ont une coordonnée en commun.

Définissez les axiomes de la fonction *rechercher* qui recherche dans un arbre quaternaire ordonné un élément représenté par sa clé. Celle-ci est définie par ses deux coordonnées de longitude et de latitude.

Exercice 20.11. Écrivez les méthodes `ajouter` et `supprimer` pour une table représentée par un arbre 2-3-4. Un conseil : écrivez la méthode `supprimer` de façon itérative, et pensez à mémoriser tous les nœuds traversés, pendant la recherche de la clé à supprimer.

Exercice 20.12. Modifiez la méthode `rechercher` dans une table implantée à l'aide d'un arbre 2-3-4, afin qu'elle retourne tous les éléments de même clé.

Exercice 20.13. Donnez l'algorithme de la suppression d'un élément dans une table d'adressage dispersé qui utilise des fonctions secondaires, d'une part, cas le cas de la méthode linéaire, d'autre part dans le cas de la méthode quadratique.

Exercice 20.14. Lorsque le taux de remplissage d'une table d'adressage dispersé ouvert devient trop important, un prochain ajout peut considérer la table pleine. Il est alors possible de réorganiser (en anglais *rehashing*) les éléments dans une nouvelle table plus grande (*e.g.* deux fois la taille de la table initiale). Modifiez l'algorithme de l'opération *ajouter* pour mettre en œuvre cette technique lorsque le taux de remplissage dépasse une certaine valeur.

Exercice 20.15. Les correcteurs orthographiques utilisent des tables d'adressage dispersé pour conserver les mots d'un dictionnaire. Une technique utilisée pour réduire l'encombrement en mémoire du dictionnaire est de ne conserver qu'une table de bits, initialisée à 0, et telle que pour tout mot m du dictionnaire, on calcule $t[h(m)] \leftarrow 1$.

Combien d'éléments doit posséder cette table ? Quelle est la réduction de place en mémoire obtenue ?

Si pour un mot m' quelconque, $t[h(m')] = 0$, que peut-on en déduire ? Et que peut-on déduire si $t[h(m')] = 1$? Quelle est la probabilité d'erreur pour une table de longueur n ?

Programmez un petit correcteur orthographique avec cette méthode. Il est possible d'améliorer la méthode en calculant plusieurs fonctions d'adressage indépendantes. On teste la présence d'un mot dans la table en vérifiant que tous les $t[h_i(m')]$ sont égaux à 1.

Chapitre 21

Files avec priorité

Les files avec priorité remettent en question le modèle FIFO des files ordinaires présentées au chapitre 17. Avec ces files, l'ordre d'arrivée des éléments n'est plus respecté. Les éléments sont munis d'une priorité et ceux qui possèdent les priorités les plus fortes sont traités en premier. Les systèmes d'exploitation utilisent fréquemment les files avec priorité, par exemple, pour gérer l'accès des travaux d'impression à une imprimante, ou encore l'accès des processus au processeur.

Dans ce chapitre, après avoir décrit le type abstrait des files avec priorité, nous présenterons deux mises en œuvre possibles : une première à l'aide de liste, et une seconde, très efficace, à l'aide de tas.

21.1 DÉFINITION ABSTRAITE

Le type abstrait $\mathcal{F}ile{-}\mathcal{P}riorité$ définit l'ensemble des files avec priorité. Les éléments de ces files appartiennent à l'ensemble $\mathcal{E}$ et sont munis d'une priorité prise dans $\mathcal{P}riorité$. L'ensemble $\mathcal{P}riorité$ est pourvu d'une relation d'ordre total qui permet d'ordonner les éléments du plus prioritaire au moins prioritaire. Les opérations suivantes sont définies sur le type abstrait $\mathcal{F}ile{-}\mathcal{P}riorité$:

est-vide?	:	$\mathcal{F}ile{-}\mathcal{P}riorité$	$\rightarrow$	booléen
ajouter	:	$\mathcal{F}ile{-}\mathcal{P}riorité \times \mathcal{E} \times \mathcal{P}riorité$	$\rightarrow$	$\mathcal{F}ile{-}\mathcal{P}riorité$
premier	:	$\mathcal{F}ile{-}\mathcal{P}riorité$	$\rightarrow$	$\mathcal{E}$
supprimer	:	$\mathcal{F}ile{-}\mathcal{P}riorité$	$\rightarrow$	$\mathcal{F}ile{-}\mathcal{P}riorité$

L'opération *est-vide?* teste si la file est vide ou pas, l'opération *ajouter* insère dans la file un nouvel élément avec sa priorité, l'opération *premier* retourne l'élément le plus prioritaire

de la file, et enfin l'opération *supprimer* retire l'élément le plus prioritaire de la file. Les axiomes du type abstrait sont laissés en exercices.

21.2 REPRÉSENTATION AVEC UNE LISTE

Une représentation qui vient en premier à l'esprit pour le type abstrait $\mathcal{F}ile\text{–}\mathcal{P}riorité$ est la structure de liste. Les listes permettent une mise en œuvre très simple de ce type abstrait. Dans cette section, nous ne nous intéresserons qu'à la complexité des opérations du type abstrait, selon que la liste est *ordonnée* ou *non*. L'écriture des algorithmes et la programmation en JAVA de ces opérations ne posent guère de difficultés et nous les laisserons en exercice.

L'ajout d'un nouvel élément dans une liste non ordonnée se fait en tête de liste. Le coût de l'opération est efficace, il est en $\mathcal{O}(1)$. En revanche, comme les éléments sont dans un ordre quelconque, les opérations *premier* et *supprimer* nécessitent une recherche linéaire de l'élément le plus prioritaire. Cette recherche peut demander un parcours complet de la liste, et la complexité de ces deux opérations est $\mathcal{O}(n)$.

Lorsque le type abstrait $\mathcal{F}ile\text{–}\mathcal{P}riorité$ est implanté à l'aide une liste ordonnée, les opérations *premier* et *supprimer* sont en $\mathcal{O}(1)$, puisqu'on fera en sorte que les éléments de la liste soient placés par ordre de priorité décroissante. L'opération *ajouter* doit maintenir l'ordre sur les éléments de la liste, et sa complexité est celle d'une recherche dans une liste ordonnée, c'est-à-dire $\mathcal{O}(n)$.

Le choix de la représentation d'une file avec priorité sera dictée par la nature des opérations utilisées. S'il y a peu d'ajouts et beaucoup de consultations du premier de la file, on préférera une liste ordonnée. En revanche, s'il y a de nombreux ajouts, en alternance avec des suppressions qui maintiennent une file de petite taille, une liste non ordonnée sera très efficace.

Quoi qu'il en soit, la représentation du type abstrait $\mathcal{F}ile\text{–}\mathcal{P}riorité$ à l'aide de listes linéaires n'est pas très efficace, surtout si les files possèdent un nombre d'éléments significatif, c'est-à-dire au delà d'une vingtaine d'éléments. Pour les files de grandes tailles, il existe une représentation très efficace, appelée *tas*, que nous allons décrire maintenant.

21.3 REPRÉSENTATION AVEC UN TAS

Un *tas* est un arbre binaire parfait partiellement ordonné. Rappelons qu'un arbre parfait est un arbre dont toutes les feuilles sont situées sur au plus deux niveaux. Les feuilles du dernier niveau sont placées le plus à gauche (voir le chapitre 19 page 229).

La relation d'ordre que définit un arbre partiellement ordonné sur ses éléments est telle que la valeur de la racine est supérieure ou égale à la valeur des nœuds de ses sous-arbres gauche et droit. La valeur la plus grande d'un tel arbre est toujours située à la racine. Notez que la relation ordre aurait pu tout aussi bien être inversée, de telle façon que la valeur la plus petite soit à la racine. La figure 21.1 montre un exemple d'arbre partiellement ordonné.

Un tas réunit à la fois les propriétés d'un arbre binaire parfait, et d'un arbre partiellement ordonné. La figure 21.2 montre l'arbre de la figure 21.1 organisé en tas.

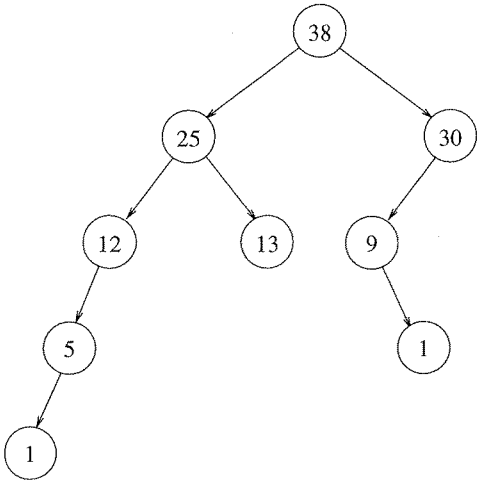


Figure 21.1 – *Un arbre partiellement ordonné.*

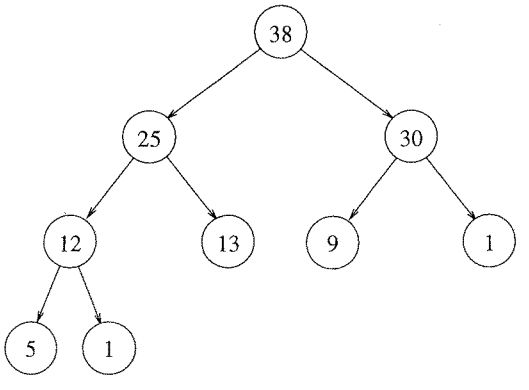


Figure 21.2 – *L'arbre de la figure 21.1 sous forme de tas.*

Un tas offre une représentation très efficace du type abstrait $\mathcal{F}ile\text{–}\mathcal{P}riorité$, puisqu'avec une telle structure, la complexité des opérations est $\mathcal{O}(1)$ ou $\mathcal{O}(\log_2 n)$.

Par la suite, et pour simplifier, seules les valeurs des priorités apparaîtront dans les figures. Ces priorités sont des entiers, et plus l'entier sera grand, plus la priorité sera forte.

21.3.1 Premier

Puisque l'élément le plus prioritaire est toujours placé à la racine de l'arbre, l'accès à sa valeur est très efficace, et sera toujours en $\mathcal{O}(1)$.

Algorithme `premier(t)`

```
{Rôle : retourne l'élément le plus prioritaire du tas t}
rendre racine(t)
```

Nous allons voir maintenant que les opérations d'ajout et de suppression d'éléments, qui demandent une réorganisation du tas, sont elles aussi performantes. Elles ne nécessitent que le parcours d'une branche de l'arbre binaire. Comme l'arbre est parfait, la hauteur d'une branche est égale au plus à $\lfloor \log_2 n \rfloor$, où n est le nombre d'éléments dans le tas. La complexité des opérations d'ajout et de suppression est, au pire, égale à $\mathcal{O}(\log_2 n)$.

21.3.2 Ajouter

L'opération d'ajout consiste, dans un premier temps, à placer le nouvel élément et sa priorité en feuille, celle qui suit la dernière feuille de l'arbre. Le nouvel élément n'est pas nécessairement correctement placé et l'ordre partiel sur les éléments du tas n'est peut-être plus respecté. Il faut alors réordonner le tas, et chercher une *bonne* place pour le nouvel élément.

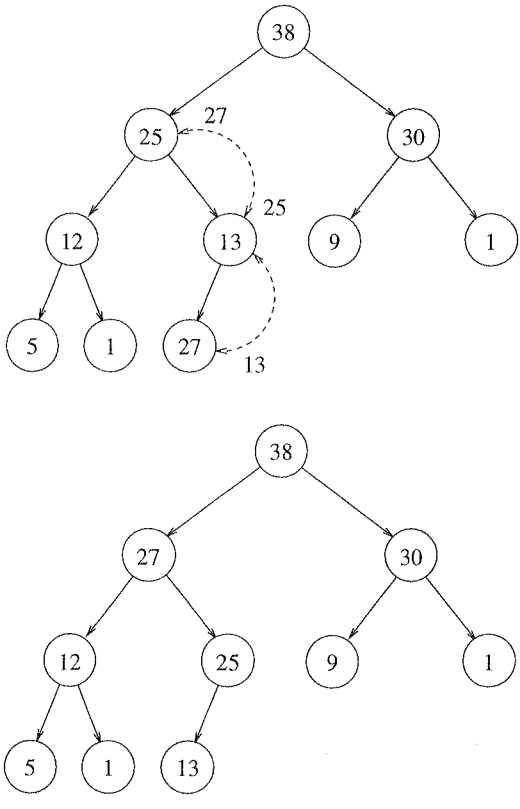


Figure 21.3 – *Ajout d'un élément de priorité 27.*

Algorithme `ajouter(t, e, p)`

```
a ← cons((e,p),∅,∅)
mettre a après la dernière feuille du tas t
réordonner-tas1(t,a)
```

L'algorithme de réordonnancement du tas procède par échanges successifs de la valeur du nouvel élément avec la valeur de ses pères sur la branche qui remonte jusqu'à la racine du tas. Tant que sa valeur de priorité est supérieure à celle du père, on fait l'échange. L'algorithme s'arrête quand un père possède une valeur de priorité supérieure ou égale, ou bien lorsque la racine du tas est atteinte. Dans ce dernier cas, le nouvel élément est placé à la racine du tas. La figure 21.3 page 288 montre l'ajout d'un élément de priorité 27. Après avoir été placé en feuille, l'élément est successivement échangé avec les éléments de priorité 13 et 25. Sa position finale est celle du fils gauche de la racine.

L'algorithme de réordonnancement est décrit récursivement comme suit :

```
Algorithme réordonner-tas1(t,a)
   si a≠t alors
      si priorité(racine(a)) > priorité(racine(père(a))) alors
           échanger(valeur(racine(a)),valeur(racine(père(a))))
           réordonner-tas1(t,père(a))
      finsi
   finsi
```

21.3.3 Supprimer

La suppression de l'élément le plus prioritaire du tas consiste à remplacer la valeur de la racine du tas par la valeur de la dernière feuille du tas, puis à supprimer cette feuille, et enfin à réordonner le tas.

```
Algorithme supprimer(t)
   racine(t) ← racine(dernière feuille du tas t)
   supprimer la dernière feuille de t
   réordonner-tas2(t)
```

Comme précédemment, la réorganisation du tas procède par échanges successifs, mais en partant, cette fois-ci, de la racine et en descendant vers les feuilles. Si elle lui est inférieure, la valeur déplacée est échangée avec la valeur maximale des priorités de ses fils gauche ou droit.

La figure 21.4 page 290 montre la suppression de l'élément le plus prioritaire du tas, *i.e.* 38. L'élément de priorité 13 est supprimé et copié à la racine du tas, et sa feuille est supprimée. Il est ensuite échangé avec l'élément de priorité 30.

```
Algorithme réordonner-tas2(a)
   si a n'est pas une feuille alors
     {chercher le fils qui possède la valeur min}
     si ∄ sad(a) ou priorité(racine(sag(a))>priorité(racine(sad(a)))
       alors
            {un seul fils gauche ou
             le fils gauche a la priorité maximale}
            fils-max ← sag(a)
       sinon {le fils droit a la priorité maximale}
            fils-max ← sad(a)
   finsi
```

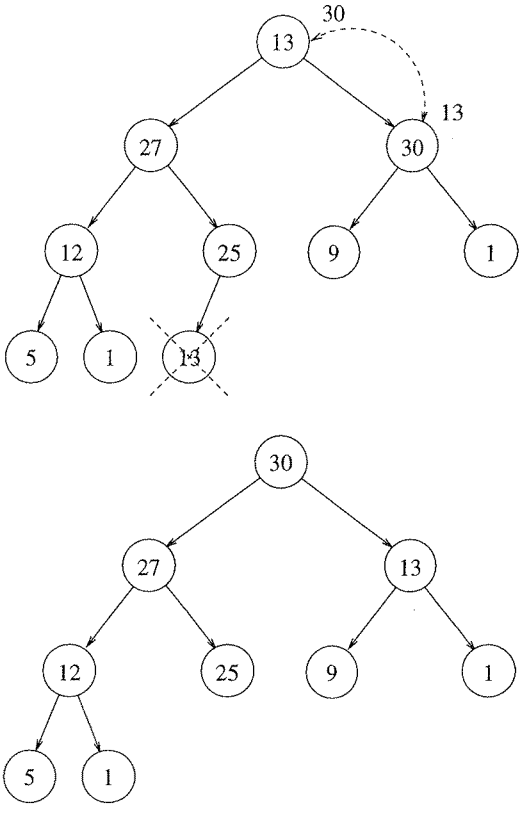


Figure 21.4 – *Suppression d'un élément le plus prioritaire.*

```
        {échanger la valeur de a avec fils-min si nécessaire}
        si priorité(racine(a))<priorité(racine(fils-max)) alors
           {racine(i) a la priorité maximale}
           échanger(valeur(racine(a)),valeur(racine(fils-max)))
           réordonner-tas2(fils-max)
        finsi
     finsi
```

21.3.4 L'implantation en Java

Les signatures des opérations du type abstrait $\mathcal{F}ile\text{–}\mathcal{P}riorité$ sont données par l'interface `FilePriorité` :

```
public interface FilePriorité {
    public boolean estVide();
    public Object premier() throws FileVideException;
    public void supprimer() throws FileVideException;
    public void ajouter(Object e, Object p);
}
```

L'utilisation de tableau pour implanter les tas est particulièrement efficace car l'accès aux nœuds de l'arbre est direct. De plus, nous l'avons déjà vu, le père d'indice i est lié à ses fils gauche et droit d'indice, respectivement, $2i$ et $2i + 1$. Et le père d'un nœud d'indice i est à l'indice $i/2$. Par exemple, dans la figure 21.5, le nœud de valeur 25 est à l'indice 2, et possède deux fils 12 et 13 aux indices $2 \times 2 = 4$ et $2 \times 2 + 1 = 5$. Le père du nœud de valeur 1 et d'indice 9, est à l'indice $9/2 = 4$.

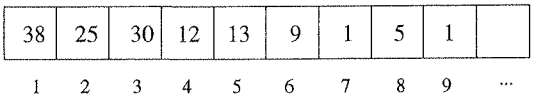


Figure 21.5 – *Le tas de la figure 21.2 dans un tableau.*

On définit la classe `TasListeTableau` qui implante l'interface `FilePriorité`, et qui étend la classe `ListeTableau` représentant les listes mises en œuvre à l'aide d'un tableau. L'arithmétique sur les indices est alors faite sur les rangs des éléments dans la liste. Remarquez que nous aurions pu étendre n'importe quelle autre classe d'implantation du type abstrait *Liste*, mais pour des raisons d'efficacité en terme d'accès aux éléments de la liste, seule la classe `ListeTableau` convient.

L'attribut `cmp` conserve les opérations de comparaisons nécessaires pour établir l'ordre partiel sur les éléments du tas.

```
public class TasListeTableau extends ListeTableau
implements FilePriorité {
    protected Comparable cmp;
    ...
}
```

Les valeurs des éléments de la file et leurs priorités seront conservées dans des objets de type `Élément`, défini à la page 241. L'attribut `clé` représente la priorité de l'élément. L'élément qui possède la clé de valeur la plus grande possède la priorité la plus forte.

Afin d'accroître la lisibilité des méthodes qui implantent les opérations du type abstrait *File–Priorité*, nous définissons les méthodes privées suivantes, qui retournent, respectivement, la valeur (de type `Élément`) du nœud courant, du nœud de son père et de ses fils gauche et droit.

```
// Rôle: retourne la valeur du nœud d'indice i
private Élément racine(int i) {
    return ((Élément) ième(i));
}
// Rôle: retourne la valeur du nœud du père du nœud d'indice i
private Élément père(int i) {
    return ((Élément) ième(i/2));
}
// Rôle: retourne la valeur du nœud du sag du nœud d'indice i
private Élément sag(int i) {
    return ((Élément) ième(2*i));
}
```

```
// Rôle: retourne la valeur du nœud du sad du nœud d'indice i
private Élément sad(int i) {
     return ((Élément) ième(2*i+1));
}
```

La méthode `premier` retourne la valeur de la racine du tas. La racine est au premier rang de la liste.

```
public Object premier() throws FileVideException {
     if (estVide()) throw new FileVideException();
     return racine(1).valeur();
}
```

Le nouvel élément est ajouté en feuille à l'extrémité du tas. Son rang dans la liste est égal à `longueur()+1`. L'ajout de la nouvelle feuille dans la liste est fait par la méthode `ajouter` de la super classe `ListeTableau`. L'algorithme récursif de réordonnancement donné plus haut s'écrit simplement et efficacement de façon itérative.

```
public void ajouter(Object e, Object p) {
   int i=longueur();
   // mettre le nouvel élément à la dernière place
   ajouter(++i, new Élément(e,p));
   // réordonnancement du tas
   while (i>1 && cmp.supérieur(racine(i).clé(), père(i).clé()))
            échanger(i, i/=2);
}
```

Pour supprimer l'élément prioritaire de la file, on affecte au nœud de la racine, la valeur du nœud de rang `longueur()`, et on supprime ce dernier élément de la liste. Le réordonnancement du tas s'écrit également de façon itérative, et traite, en partant de la racine, les nœuds qui ne sont pas des feuilles (c'est-à-dire les nœuds de rang 1 à `longueur()/2` jusqu'à ce que la position finale soit trouvée.

```
public void supprimer() {
   // changer la valeur de la racine
   affecter(1,ième(longueur()));
   // supprimer la dernière feuille
   supprimer(longueur());
   int i=1, lg=longueur();
   // réordonnancement du tas
   while (i<=lg/2) {
     // le nœud i possède au moins un fils
      int filsMax =
      (2*i == lg || cmp.supérieur(sag(i).clé(),sad(i).clé())) ?
      // un seul fils gauche ou le fils gauche a la priorité maximale
      2*i
      // le fils droit a la priorité maximale
      : 2*i+1;
      if (cmp.inférieur(racine(i).clé(), racine(filsMax).clé()))
      {
         // racine(i) a la priorité maximale
```

```
            échanger(i,filsMax);
            i = filsMax;
        }
        else // racine(i) est à sa position finale
            return;
    }
    // racine(i) est à sa position finale
} // fin supprimer
```

21.4 EXERCICES

Exercice 21.1. Proposez un algorithme en $\mathcal{O}(n \log_2 n)$ qui trouve la k^{e} plus grande valeur d'une suite quelconque d'entiers lus sur l'entrée standard.

Exercice 21.2. Proposez un algorithme de recherche d'un élément dans un tas. Quelle est sa complexité ? Est-ce qu'un tas est adapté à ce type d'opération ?

Exercice 21.3. Programmez une implantation de l'interface `FilePriorité` à l'aide d'un tas mis en œuvre avec une structure chaînée.

Exercice 21.4. Une généralisation des tas consiste à les représenter par des arbres quelconques. L'arbre est toujours partiellement ordonné, mais le nombre de fils de chaque nœud n'est plus limité à deux. Donnez les algorithmes des opérations *ajouter* et *supprimer* pour un tas dont le nombre fils de chaque nœud est borné par m. Indiquez la complexité de ces opérations, puis comparez les avantages et les inconvénients d'un tas binaire et d'un tas d'ordre m.

Chapitre **22**

Algorithmes de tri

22.1 INTRODUCTION

Un tri consiste à ordonner de façon croissante (ou décroissante) une suite d'éléments à partir des clés qui leur sont associées. Pour une suite de n éléments, il s'agit donc de trouver une permutation particulière des éléments parmi les $n!$ permutations possibles. Les méthodes de tri sont connues depuis fort longtemps, et leurs algorithmes ont été étudiés en détail, notamment dans [Knu73].

Prenons, par exemple, la suite [1] d'entiers à ordonner suivante :

```
53   914   230   785   121   350   567   631    11   827   180
```

Une opération de tri ordonnera les éléments de façon croissante et retournera la suite :

```
11    53   121   180   230   350   567   631   785   827   914
```

On distingue les tris *internes* des tris *externes*. Un tri est dit interne, si l'ensemble des clés à trier réside en mémoire principale. Les éléments à trier sont généralement placés dans des tableaux. Au contraire, un tri est externe lorsque l'ensemble des clés ne peut résider en mémoire centrale, et doit être conservé dans des fichiers. Ces tris utilisent des méthodes d'interclassement et cherchent à minimiser le nombre de fichiers auxiliaires utilisés.

Dans l'exemple précédent, la clé est une simple valeur entière. Toutefois, les clés peuvent être structurées, et posséder plusieurs *niveaux*. On parle alors de clés *primaire*, *secondaire*, etc. Dans un annuaire téléphonique, les abonnés sont classés par ordre alphabétique d'abord

1. Dans tout ce chapitre, les algorithmes présentés seront appliqués sur cette suite de référence, et pour simplifier nous assimilerons la valeur de l'élément à sa clé.

sur les noms de famille (la clé primaire), puis en cas d'homonymie, par ordre alphabétique sur les prénoms (la clé secondaire). La complexité de la clé ne modifie pas les algorithmes de tri, seule la méthode de comparaison des éléments change.

22.2 TRIS INTERNES

Une opération de tri interne est définie par la signature suivante :

$$\text{trier} \quad : \quad \mathcal{L}iste \quad \rightarrow \quad \mathcal{L}iste$$

Nous considérerons que les éléments de la liste l retournée par l'opération de tri sont en ordre croissant tel que :

$$\forall i,j \in [1, \text{longueur}(l)],\ i < j \Rightarrow \text{clé}(\text{ième}(l,i)) \leqslant \text{clé}(\text{ième}(l,j))$$

Un tri peut se faire *sur place*, c'est-à-dire qu'une seule liste est utilisée, ou *avec recopie*, il a alors recours à une ou plusieurs listes auxiliaires (en général une seule). Dans cette section, nous ne présenterons que des méthodes de tri sur place par *comparaison* de clés.

La complexité temporelle des algorithmes de tri s'exprime en nombre de *comparaisons* des clés. Selon l'ordre initial des clés dans la liste, la complexité d'un tri pourra varier. Nous distinguerons la complexité moyenne, la meilleure et la pire. En général, on se réfère à la complexité moyenne pour établir les performances d'un tri. Il est démontré [Knu73] que la complexité la pire d'un tri sur place par comparaison de clés *ne peut être inférieure à* $\mathcal{O}(n \log_2 n)$.

Le nombre de comparaisons n'est toutefois pas suffisant pour décrire totalement l'efficacité d'un tri. La complexité en terme de nombre de *déplacements* des éléments a une incidence non négligeable sur les performances des tris. Le coût d'une opération d'affectation d'élément, surtout si sa taille est grande, peut être supérieur à celui d'une opération de comparaison. Selon les méthodes de tris, nous évaluerons les déplacements ou les échanges.

Les tris internes peuvent être classés en deux catégories selon leur complexité. D'une part, les tris *simples* dont la complexité moyenne en comparaisons est quadratique, et qui ont des performances médiocres. D'autre part, des tris *élaborés* dont les algorithmes sont plus complexes, mais plus performants puisque leur complexité moyenne en comparaisons est logarithmique.

Une seconde classification possible peut être faite selon la méthode de tri utilisée. Dans cette section, nous présenterons trois catégories de méthode de tri interne sur place : par *sélection*, par *insertion*, et par *échanges*. Pour chacune de ces catégories, nous donnerons un tri simple et un tri élaboré.

22.2.1 L'implantation en Java

L'implantation des algorithmes de tri utilisera le type `Liste` donné au chapitre 17, complété par les méthodes `affecter` et `échanger`. Ces opérations sont définies par les signatures et

les axiomes suivants :

$$\begin{array}{lll} \text{affecter} & : & \mathcal{L}iste \times \text{entier} \times \mathcal{E} \rightarrow \mathcal{L}iste \\ \text{échanger} & : & \mathcal{L}iste \times \text{entier} \times \text{entier} \rightarrow \mathcal{L}iste \end{array}$$

$\forall l \in \mathcal{L}iste, \forall i,j \in [1, \text{longueur}(l)], \forall e \in \mathcal{E}$

(1) $\text{longueur}(\text{échanger}(l,i,j)) = \text{longueur}(l)$

(2) $\text{ième}(\text{échanger}(l,i,j),i) = \text{ième}(l,j)$

(3) $\text{ième}(\text{échanger}(l,i,j),j) = \text{ième}(l,i)$

(4) $\text{longueur}(\text{affecter}(l,i,e)) = \text{longueur}(l)$

(5) $\forall r \in [1, \text{longueur}(l)]$ et $r \neq i$, $\text{ième}(\text{affecter}(l,i,e),r) = \text{ième}(r)$

(6) $r = i, \text{ième}(\text{affecter}(l,i,e),r) = e$

Afin que ces algorithmes soient efficaces, l'implantation de la liste devra employer un tableau pour un accès direct aux éléments de la liste. Les éléments à trier sont de type `Élément` défini à la page 241. Les opérations de comparaisons spécifiques au type des clés des éléments sont passées en paramètre de la méthode de tri. Les en-têtes des méthodes de tri auront la forme suivante :

```
public void tri(Liste l, Comparable c);
```

22.2.2 Méthodes par sélection

Le principe des méthodes de tri par sélection est de rechercher le minimum de la liste, de le placer en tête et de recommencer sur le reste de la liste. À la i^e étape, la sous-liste formée des éléments du rang 1 au rang $i-1$ est triée, et tous les éléments du rang i au rang n possèdent des clés supérieures ou égales à celles des éléments de la sous-liste déjà triée. L'élément de clé minimale, trouvée entre le rang i et le rang n, est placé au rang i (voir la figure 22.1), et le tri se poursuit à l'étape $i+1$.

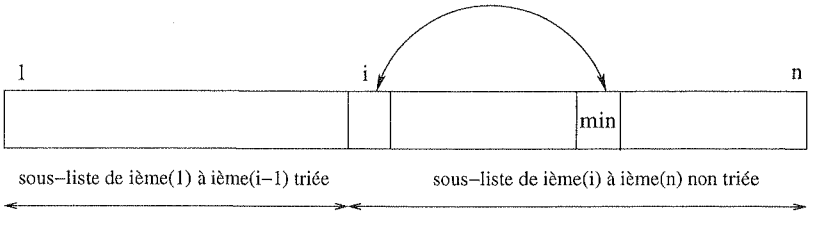


Figure 22.1 – *Tri par sélection à l'étape i.*

Nous présentons deux tris par sélection. Le premier, le tri par sélection directe, a des performances médiocres, sa complexité est $\mathcal{O}(n^2)$. Le second, le tri en tas, est particulièrement efficace. Sa complexité est au pire égale à $\mathcal{O}(n \log_2 n)$.

Sélection directe

Dans cette méthode de tri, la recherche du minimum à partir de l'élément de rang i, est une simple recherche linéaire. Le déroulement de cette méthode, en utilisant la suite d'entiers de référence, se déroule comme suit :

```
|53   914  230  785  121  350  567  631   11  827  180
 11  |914  230  785  121  350  567  631   53  827  180
 11    53 |230  785  121  350  567  631  914  827  180
 11    53  121 |785  230  350  567  631  914  827  180
 11    53  121  180 |230  350  567  631  914  827  785
 11    53  121  180  230 |350  567  631  914  827  785
 11    53  121  180  230  350 |567  631  914  827  785
 11    53  121  180  230  350  567 |631  914  827  785
 11    53  121  180  230  350  567  631 |914  827  785
 11    53  121  180  230  350  567  631  785 |827  914
```

À la i^e étape, la clé courante est à droite de la barre verticale, et le minimum de la sous-liste est souligné. Cet algorithme de tri s'écrit comme suit :

```
Algorithme SélectionDirecte(l)
   {Rôle: trie la liste l en ordre croissant des clés}
   pourtout i de 1 à longueur(l)-1 faire
      {Invariant : la sous-liste de ième(l) à ième(i-1) est triée et
       ∀k∈[1,i-1],∀k' ∈[i,longueur(l)],clé(ième(l,k))≤clé(ième(l,k'))}
      min ← i
      {chercher l'indice du min entre ième(i) et ième(longueur(l))}
      pourtout j de i+1 à longueur(l) faire
         si clé(ième(l,j))<clé(ième(l,min)) alors
               min ← j
         finsi
      finpour
      {échanger ième(l,i) avec ième(l,min)}
      si i≠min alors ième(l,i) ↔ ième(l,min) finsi
   finpour
```

Pour une liste de longueur n, la première boucle est exécutée $n-1$ fois. À l'itération i, la boucle interne produit $n-i$ itérations. Puisque la comparaison est exécutée systématiquement, le nombre de comparaisons moyen, le pire et le meilleur sont tous les trois égaux à $\sum_{i=1}^{n-1} i = \frac{1}{2}(n^2-n) = \mathcal{O}(n^2)$.

Il y a un échange à chaque itération, sauf dans le cas où le minimum est déjà au rang i. Si on considère que la probabilité est $1/(n-i)$ pour que le minimum soit au rang i, le tri complet produit donc $n-1-\sum_{i=2}^{n} 1/i \approx n - \ln(n)$ échanges, soit une complexité égale à $\mathcal{O}(n)$.

Tri en tas

Le tri en tas, ou *heapsort* en anglais, est une méthode de tri par sélection particulièrement efficace puisque sa complexité est au pire égale à $\mathcal{O}(n \log_2 n)$.

Cette méthode organise la partie de la liste qui n'est pas encore triée en tas (voir la section 21.3, à la page 286). Les éléments les plus prioritaires possèdent les clés les plus grandes. Pour des raisons d'implémentation (nous avons vu qu'il est commode que la racine du tas soit au rang 1), le tas est placé en tête de liste. Au départ, avant l'opération de tri proprement dite, l'algorithme doit procéder à la création d'un tas initial, à partir de tous les éléments de la liste. Nous décrirons cette première phase un peu plus loin.

Voyons comment procède ce tri à la i^e étape. À ce moment du tri, la liste est organisée comme l'indique la figure 22.2. Tous les éléments compris entre le rang $i+1$ et n sont ordonnés de façon croissante et possèdent des clés supérieures ou égales à celles des éléments du tas, compris entre le rang 1 et i. L'élément de rang 1 possède la clé la plus grande du tas.

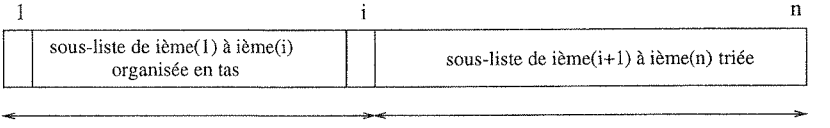


Figure 22.2 – *Tri en tas à l'étape i.*

Le prochain élément à placer au rang i est l'élément dont la clé est la plus grande sur l'intervalle $[1,i]$. Sa recherche dans le tas est immédiate puisque c'est le premier de la liste. La complexité de la recherche est donc égale à $\mathcal{O}(1)$. L'élément de rang i prend ensuite la place du premier élément de la liste qui est mémorisé. Le tas, formé des éléments compris entre les rangs 1 et $i-1$, est alors réordonné par l'algorithme donné à la page 289. Enfin, l'élément de clé maximale mémorisé est placé au rang i. L'algorithme complet du tri en tas est le suivant :

```
Algorithme triEnTas(l)
   {Rôle: trie la liste l en ordre croissant des clés}
   création du tas initial
   {la liste l est organisée en tas}
   pourtout i de longueur(l) à 2 faire
      {Invariant : la sous-liste de ième(i+1) à ième(l.longueur())}
      {est triée et ∀k∈[1,i-1],∀k' ∈[i,longueur(l)],
       clé(ième(l,k))≤clé(ième(l,k'))}
      max = ième(l, 1)
      ième(l,1) ← ième(l,i)
      {réordonner le tas entre 1 et i-1}
      réordonner-tas2(l, 1, i-1)
      {max est le prochain ième(i)}
      ième(l,i) ← max
   finpour
```

L'étape initiale de création du tas à partir de la liste de référence construit la liste :

```
914 827  567  785  180  350  230  631   11  121   53
```

Le déroulement de la méthode de tri en tas, appliquée à cette liste produit les dix étapes suivantes :

914	827	567	785	180	350	230	631	11	121	|53
827	785	567	631	180	350	230	53	11	|121	914
785	631	567	121	180	350	230	53	|11	827	914
631	180	567	121	11	350	230	|53	785	827	914
567	180	350	121	11	53	|230	631	785	827	914
350	180	230	121	11	|53	567	631	785	827	914
230	180	53	121	|11	350	567	631	785	827	914
180	121	53	|11	230	350	567	631	785	827	914
121	11	|53	180	230	350	567	631	785	827	914
53	|11	121	180	230	350	567	631	785	827	914
11	53	121	180	230	350	567	631	785	827	914

Pour construire le tas initial, il faut ordonner de façon partielle tous les éléments de la liste. Chaque élément correspond à un nœud particulier d'un arbre parfait. On applique l'algorithme de réordonnancement `réordonner-tas2` sur chacun des nœuds (qui ne sont pas des feuilles), en partant des nœuds de rang $longueur(l)/2$ jusqu'à la racine, c'est-à-dire le nœud de rang 1. Pour chaque nœud i, les éléments compris entre les rangs i et $longueur(l)$ sont réordonnés en tas. Le tas initial est construit comme suit :

```
Algorithme création du tas
   pourtout i de longueur(l)/2 à 1 faire
        réordonner-tas2(l, i, longueur(l) )
   finpour
```

Écriture en JAVA

Nous donnons, ci-dessous, l'écriture complète du tri en tas en JAVA. La méthode `réordonnerTas` réordonne le tas depuis le rang i jusqu'au rang n, en fonction de l'élément de rang i, Le paramètre `c` donne les opérations de comparaisons sur les clés des éléments. Les méthodes `racine`, `sag` et `sad` sont celles données à la page 291.

```
private void réordonnerTas(Liste l, int i, int n, Comparable c) {
  while (i<=n/2) {
     // le nœud i possède au moins un fils
     int filsMax = (2*i == n
                  || c.supérieur(sag(l,i).clé(), sad(l,i).clé())) ?
          // un seul fils gauche ou le fils gauche
          // a la priorité maximale
          2*i
          // le fils droit a la priorité maximale
          : 2*i+1;
     if (c.inférieur(racine(l,i).clé(),racine(l,filsMax).clé())) {
          // le nœud i a la priorité maximale
          l.échanger(i,filsMax);
          i = filsMax;
     }
```

```
      else return;
    }
  } // fin réordonnerTas
```

Enfin, la méthode principale du tri en tas est une traduction directe de l'algorithme. Elle mémorise dans une variable temporaire `max` le maximum du tas situé au rang 1 de la liste. Celui-ci est remplacé par la valeur de l'élément de rang i. Le tas est alors diminué d'un élément, et il est réordonné entre les rangs 1 et $i-1$. Enfin, la valeur de `max` est affectée au rang i, sa place définitive.

```
public void triEnTas(Liste l, Comparable c) {
  // construction du tas initial
  for (int i=l.longueur()/2; i>=1; i--)
      réordonnerTas(l,i,l.longueur(),c);
  // tri de la liste
  for (int i=l.longueur(); i>1; i--) {
       // invariant: la sous-liste de ième(i+1) à ième(l.longueur())
       // est triée et ∀k∈[1,i-1],∀k'∈[i,longueur(l)],
       // clé(ième(l,k))<=clé(ième(l,k'))
     Object max=l.ième(1);
     l.affecter(l,l.ième(i));
     // réordonner le tas entre 1 et i-1
     réordonnerTas(l,1,i-1,c);
     // max est le prochain ième(i)
     l.affecter(i,max);
  }
}
```

Complexité du tri en tas

La complexité du tri en tas est égale à la complexité de la création du tas initial, plus celle du tri proprement dit. Nous allons nous intéresser à la complexité dans le pire des cas.

La complexité de la création du tas est égale à la somme des coûts de réordonnancement de chacun des sous-arbres du tas. Puisque chaque élément du tas est la racine d'un sous-arbre parfait, il y a n sous-arbres à réordonner. On définit le coût total du réordonnancement comme la somme des coûts des réordonnancements de chacun des sous-arbres, exprimé en nombre d'itérations effectuées dans la méthode `réordonnerTas`.

Le coût par nœud est borné par la profondeur du sous-arbre dont il est la racine. Pour les nœuds de rang compris entre $n/2+1$ et n, le coût est nul puisque ce sont des feuilles. Pour les nœuds de rang compris entre $n/4+1$ et $n/2$, le coût est inférieur ou égal à 1. Pour les nœuds de rang compris entre $n/8+1$ et $n/4$, le coût est inférieur ou égal à 2. Et ainsi de suite jusqu'à la racine du tas, où le coût est inférieur ou égal à $\log_2(n+1)-1$. Le coût maximal est calculé pour un arbre parfait complet, c'est-à-dire dont le dernier niveau possède toutes ses feuilles. On voit qu'il est égal à la somme :

$$S = \sum_{i=1}^{\log_2(n+1)} 2^{i-1}\log_2(n+1) - i = n - \log_2(n+1)$$

La somme S étant bornée par n, la complexité de la construction du tas est au pire $\mathcal{O}(n)$.

La phase de tri proprement dite comporte $n-1$ étapes. La complexité de chaque étape étant celle de la complexité d'un réordonnancement du tas, sa complexité est au pire égale à $\mathcal{O}(n \log_2 n)$.

En ajoutant la complexité de la création du tas initial et celle de la phase de tri, l'algorithme de tri en tas conserve une complexité, dans le pire des cas, égale à $\mathcal{O}(n \log_2 n)$.

22.2.3 Méthodes par insertion

Dans ces méthodes, à la i^e étape du tri, les $i-1$ premiers éléments forment une sous-liste triée dans laquelle il s'agit d'insérer le i^e élément à sa place (voir la figure 22.3 page 302). Nous présentons trois algorithmes selon cette méthode : le tri par insertion directe et sa variante par insertion dichotomique, et le tri par distances décroissantes.

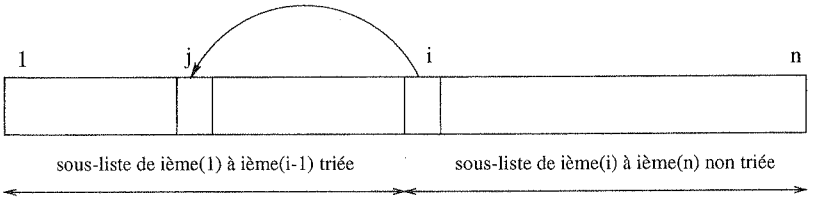


Figure 22.3 – *Tri par insertion à l'étape i.*

Insertion séquentielle

La liste est parcourue à partir du rang 2 jusqu'au dernier. À l'étape i, les $i-1$ premiers éléments forment une sous-liste triée. Un rang d'insertion du i^e élément est recherché de façon séquentielle entre le rang i et le premier rang. Il est tel que la sous-liste reste triée. Lors de l'insertion, l'élément de rang i est mémorisé dans une variable auxiliaire et un décalage d'une place vers la droite des éléments compris entre le rang j et le rang $i-1$ est nécessaire. Pour améliorer l'efficacité de l'algorithme, ce décalage doit être fait pendant la recherche du rang d'insertion.

Le tri par insertion appliqué à la liste de référence produit les étapes suivantes :

```
53 |914  230  785  121  350  567  631   11  827  180
53  914 |230  785  121  350  567  631   11  827  180
53  230  914 |785  121  350  567  631   11  827  180
53  230  785  914 |121  350  567  631   11  827  180
53  121  230  785  914 |350  567  631   11  827  180
53  121  230  350  785  914 |567  631   11  827  180
53  121  230  350  567  785  914 |631   11  827  180
53  121  230  350  567  631  785  914 |11  827  180
11   53  121  230  350  567  631  785  914 |827  180
11   53  121  230  350  567  631  785  827  914 |180
11   53  121  180  230  350  567  631  785  827  914
```

L'algorithme du tri par insertion séquentielle est le suivant :

```
Algorithme TriInsertion(l)
   {Rôle: trie la liste l en ordre croissant des clés}
   pourtout i de 2 à longueur(l) faire
      {Invariant : la sous-liste de ième(1) à ième(i-1) est triée}
      x ← ième(l,i)
      j ← i-1
      sup ← vrai
      tantque j>0 et sup faire
         {on décale et on cherche le rang d'insertion}
         {simultanément de façon séquentielle}
         si clé(ième(l,j))⩽clé(x) alors sup ← faux
         sinon {on décale}
            ième(l,j+1) ← ième(l,j)
            j ← j-1
         finsi
      fintantque
      ième(l,j+1) ← x
   finpour
```

Notez que l'opérateur logique de conjonction (&&) du langage JAVA n'évalue son second opérande booléen que si le premier est vrai. Cela permet d'éliminer la variable booléenne dans la programmation de l'algorithme.

```
public void triInsertion(Liste l, Comparable c) {
   for (int i=2; i<=l.longueur(); i++) {
      // invariant: la sous-liste de ième(1) à ième(i-1) est triée
      Élément x=(Élément) l.ième(i);
      int j=i-1;
      // rechercher le rang d'insertion
      while (j>0 && (c.supérieur(((Élément) l.ième(j)).clé(),x.clé())))
      {
         // on décale et on cherche le rang d'insertion
         // simultanément de façon séquentielle
         l.affecter(j+1,l.ième(j));
         j--;
      }
      // j+1 est ce rang d'insertion
      l.affecter(j+1,x);
   }
}
```

En utilisant une sentinelle, il est également possible de supprimer le test sur le rang inférieur de la liste (*i.e.* `j>0`) nécessaire lorsque l'insertion a lieu au premier rang. En général, on choisit comme sentinelle l'élément de rang i que l'on place au début de la liste. Si la boucle atteint la sentinelle, elle s'arrêtera de fait.

Dans le pire des cas, c'est-à-dire si la liste est en ordre inversé, il y a $i-1$ comparaisons à chaque étape (lorsque `j` est égal à zéro, seul `j>0` est évalué), soit $\sum_{i=2}^{n} i-1 = \frac{1}{2}(n^2+n)$ comparaisons. Au contraire, lorsque la liste est déjà triée, le tri donne sa meilleure complexité,

puisque le nombre de comparaisons est égal à $n-1$. Enfin, le nombre moyen de comparaisons est $\frac{1}{4}(n^2+n-2)$, si les clés sont réparties de façon équiprobable. La complexité moyenne de ce tri est $\mathcal{O}(n^2)$.

À chaque étape, le nombre de déplacements est égal au nombre de comparaisons plus un. Dans le pire des cas, il est égal à $\frac{1}{2}(n^2+n-2)$, dans le meilleur à $2(n-1)$ et en moyenne $\frac{1}{4}(n^2+5n-6)$.

Insertion dichotomique

Puisque la liste dans laquelle on recherche le rang d'insertion est triée, on voit qu'il peut être avantageux de remplacer la recherche linéaire par une recherche dichotomique. À partir de l'algorithme précédent, la programmation en JAVA de cette méthode s'écrit :

```
public void triInsertionDicho(Liste l, Comparable c) {
  for (int i=2; i<=l.longueur(); i++) {
     // invariant: la sous-liste de ième(1) à ième(i-1) est triée
     Élément x = (Élément) l.ième(i);
     if (c.inférieur(x.clé(), ((Élément) l.ième(i-1)).clé())) {
       // rechercher le rang d'insertion de x
       int gauche=1, droite=i-1;
       while (gauche<droite) {
            int milieu = (gauche+droite)/2;
            if (c.inférieurOuÉgal(x.clé(),
                                  ((Élément) l.ième(milieu)).clé()))
                 droite=milieu;
            else // clé(x)>clé(ieme(l,milieu))
                 gauche=milieu+1;
       }
       // gauche est le rang d'insertion
       // décaler tous les éléments de ce rang à i-1
       for (int j=i-1; j>=gauche; j--)
                 l.ième(j+1,l.ième(j));
       // mettre l'élément clé au rang gauche
       l.affecter(gauche,x);
     }
  }
}
```

Notez que le choix de la troisième version de la recherche dichotomique, donnée à la page 247, se justifie dans la mesure où il y a plus de recherches négatives que positives.

Le nombre de comparaisons est nettement amélioré. Nous avons vu qu'une recherche négative dans une liste de longueur n demandait au plus $\lfloor \log_2 n \rfloor + 1$ comparaisons. À chaque étape du tri, le nombre de comparaisons est égal à $\lfloor \log_2(i-1) \rfloor + 2$, soit dans tous les cas au total $\sum_{i=2}^{n} \lfloor \log_2(i-1) \rfloor + 2n - 2$ comparaisons. La complexité est égale à $\mathcal{O}(n \log_2 n)$. Toutefois, cette méthode perd beaucoup de son intérêt puisque l'insertion nécessite toujours un décalage linéaire. La complexité de ce tri reste donc $\mathcal{O}(n^2)$. Cette variante est en fait à peine plus efficace que le tri par insertion séquentielle.

Le tri par distances décroissantes

Ce tri, conçu par D.L. SHELL en 1959, est une amélioration notable du tri par insertion séquentielle. L'idée de ce tri est de former à l'étape i plusieurs sous-listes d'éléments distants d'un nombre fixe de positions. Ces sous-listes sont triées par insertion. À l'étape suivante, on réduit la distance et on recommence ce procédé. À la dernière étape, la distance doit être égale à 1.

En supposant que les valeurs des distances choisies sont 5, 2 et 1, les étapes produites par cette méthode sont données ci-dessous.

```
53  914  230  785  121  350  567  631   11  827  180
53  567  230   11  121  180  914  631  785  827  350
53   11  121  180  230  567  350  631  785  827  914
11   53  121  180  230  350  567  631  785  827  914
```

À la première étape, l'algorithme trie séparément les cinq listes suivantes formées d'éléments distants de cinq positions :

```
 53  350  180
914  567
230  631
785   11
121  827
```

À seconde étape, les éléments distants de deux positions forment les deux listes suivantes, qui sont triées :

```
 53  230  121  914  785  350
567   11  180  631  827
```

Enfin, à la troisième étape tous les éléments forment la liste suivante pour un dernier tri.

```
53   11  121  180  230  567  350  631  785  827  914
```

Il est évident que cette méthode finit par trier la liste puisque dans le pire des cas tout le travail est fait par la dernière passe car la distance séparant deux clés est égale à 1, comme dans le cas d'un tri par insertion séquentielle classique. Mais alors quel est l'avantage de cette méthode par rapport celle du tri par insertion séquentielle ? Nous avons remarqué que pour une liste triée, le nombre de comparaisons pour un tri par insertion est égal à $n - 1$. Ainsi à chaque étape, le tri courant profitera des tris des étapes précédentes. Ceci tient compte du fait qu'une sous-liste triée à l'étape i reste triée aux étapes suivantes.

```
Algorithme TriShell(l)
   {Rôle: trie la liste l en ordre croissant des clés}
   distance ← longueur(l) div 2
   tantque distance>0 faire
      {trier par insertion les éléments séparés de "distance"}
      pourtout i de distance+1 à longueur(l) faire
```

```
            x ← ième(l,i)
            j ← i-distance
            sup ← vrai
            {rechercher la place de x dans la sous-liste ordonnée}
            {et décaler simultanément}
            tantque j>0 et sup faire
               si clé(ième(l,j))⩽clé(x) alors sup ← faux
               sinon
                  ième(l,j+distance) ← ième(l,j)
                  j ← j-distance
               finsi
            {insérer x à sa place}
            ième(l,j+distance) ← x
         finpour
         {réduire la taille de moitié la distance}
         distance ← distance div 2
      fintantque
```

La complexité de ce tri est relativement difficile à calculer. Sachez qu'elle dépend très fortement de la séquence de distances choisie. Trouver la suite des distances qui donne les meilleurs résultats n'est pas simple. Le choix très courant d'une suite de puissances de deux (celui de l'algorithme présenté) n'est pas très bon, car la complexité du tri est dans le pire des cas égale à $\mathcal{O}(n^2)$. Plusieurs séquences, comme 1,3,7,...,$2^k + 1$, donnent des complexités dans le pire des cas égales à $\mathcal{O}(n^{3/2})$. D. KNUTH indique une complexité égale à $\mathcal{O}(n^{1,25})$ pour la suite $h_{k-1} = 2h_k + 1$,$h_t = 1$ et $t = \lfloor \log_3 n \rfloor - 1$. Nous donnons la programmation en JAVA du tri avec cette dernière séquence calculée et conservée dans le tableau `tableDistances`.

```
public void triShell(Liste l, Comparable c) {
  // création de la séquence de distances
  int [] tableDistances =
      new int[(int) Math.floor(Math.log(l.longueur())/Math.log(3))-1];
  int h = 1;
  for (int i=0; i<tableDistances.length; i++) {
       tableDistances[i] = h;
       h = 3 * h + 1;
  }
  // tri Shell
  for (int k=tableDistances.length-1; k>=0; k--) {
      // trier par insertion les éléments séparés de « distance »
      int distance = tableDistances[k];
      for (int i=distance+1; i<=l.longueur(); i++) {
         Élément x = (Élément) l.ième(i);
         int j = i-distance;
         // rechercher la place de x dans la sous-liste ordonnée
         // et décaler simultanément
         while (j>0 && c.supérieur(((Élément)l.ième(j)).clé(),x.clé()))
         {
             l.affecter(j+distance,l.ième(j));
```

```
                j-=distance;
            }
            // insérer x à sa place
            l.affecter(j+distance,x);
        }
    }
}
```

22.2.4 Tri par échanges

Ces méthodes de tri procèdent par échanges de paires d'éléments qui ne sont pas dans le bon ordre, jusqu'à ce qu'il n'y en ait plus. Nous présentons deux tris : le plus mauvais et le meilleur des tris internes de ce chapitre.

➢ Tri à bulles (Bubble sort)

Comme pour le tri par sélection, à la i^e itération, la sous-liste formée des éléments de rang 1 à $i-1$ est triée. De plus, les clés des éléments compris entre les rangs i et n sont supérieures à celles de la sous-liste ordonnée. En partant du rang n jusqu'au rang i, on échange deux éléments consécutifs chaque fois qu'ils ne sont pas dans le bon ordre, de telle sorte que le plus petit trouve sa place au rang i. Le nom de tri à bulles reflète le fait que les éléments les plus légers (les plus petits) remontent à la surface (*i.e.* vers le début de la liste).

Au-dessous, chaque ligne correspond à une étape et donne le résultat des échanges en partant de la fin de la liste. Les nombreux échanges effectués à chacune des étapes ne sont pas indiqués faute de place. Par exemple, à la première étape 180 et 827 sont d'abord échangés, puis 11 est échangé successivement avec toutes les valeurs de 631 à 53, soit au total neuf échanges uniquement pour la première étape.

```
|53  914  230  785  121  350  567  631   11  827  180
 11 |53  914  230  785  121  350  567  631  180  827
 11   53 |121  914  230  785  180  350  567  631  827
 11   53  121 |180  914  230  785  350  567  631  827
 11   53  121  180 |230  914  350  785  567  631  827
 11   53  121  180  230 |350  914  567  785  631  827
 11   53  121  180  230  350 |567  914  631  785  827
 11   53  121  180  230  350  567 |631  914  785  827
 11   53  121  180  230  350  567  631 |785  914  827
 11   53  121  180  230  350  567  631  785 |827  914
```

Cet algorithme de tri s'écrit :

```
Algorithme TriàBulles(l)
   {Rôle: trie la liste l en ordre croissant des clés}
   pourtout i de 1 à longueur(l)-1 faire
      {Invariant : la sous-liste de ième(1) à ième(i-1) est triée}
      pourtout j de longueur(l) à i+1 faire
         si clé(ième(l,j))<clé(ième(l,j-1)) alors
```

```
                {échanger ième(l,j) avec ième(l,j-1)}
                ième(l,j) ↔ ième(l,j-1)
            finsi
        finpour
    finpour
```

Il est facile de voir que le nombre de comparaisons moyen, le meilleur et le pire, est identique à celui du tri par sélection directe, soit $\frac{1}{2}(n^2 - n) = \mathcal{O}(n^2)$.

Dans le pire des cas, c'est-à-dire quand la liste est triée en ordre inverse, les éléments sont systématiquement échangés. Il y a $n - 1$ échanges à la première étape, $n - 2$ la seconde, etc. Puisqu'il y a $n - 1$ étapes, le nombre d'échanges le pire est donc $\frac{1}{2}(n^2 - 1) = \mathcal{O}(n^2)$. En revanche, il n'y a aucun échange si la liste est déjà triée.

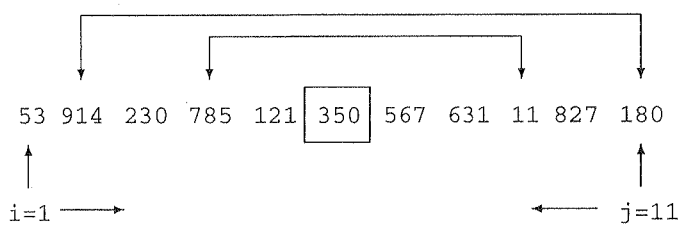
On démontre par dénombrement que le nombre d'inversions d'une liste l d'éléments distincts, c'est-à-dire le nombre de couples (i,j) tels que $i < j$ et ième$(l,i) >$ ième(l,j) est en moyenne égal à $\frac{1}{4}(n^2 - n)$. Il en résulte que c'est le nombre moyen d'échanges de l'algorithme, et sa complexité est donc $\mathcal{O}(n^2)$.

Le tri rapide

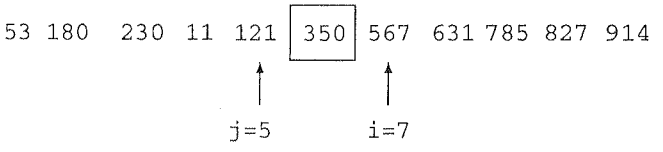
Inventé par C.A.R. HOARE au début des années soixante, le tri rapide (*quicksort* en anglais) doit son nom au fait qu'il est l'un des meilleurs tris existants. C'est un tri par échanges et *partitions*. Il consiste à choisir une clé particulière dans la liste à trier, appelée *pivot*, qui divise la liste en deux sous-listes. Tous les éléments de la première sous-liste de clé supérieure au pivot sont transférés dans la seconde. De même, tous les éléments de la seconde sous-liste de clé inférieure au pivot sont transférés dans la première. La liste est alors formée de deux partitions dont les éléments de la première possèdent des clés inférieures ou égales au pivot, et ceux de la seconde possèdent des clés supérieures ou égales au pivot. Le tri se poursuit selon le même algorithme sur les deux partitions si celles-ci possèdent une longueur supérieure à un.

À chaque étape, pour créer les deux partitions de part et d'autre du pivot, on parcourt simultanément la liste à l'aide de deux indices, en partant de ses extrémités, et on échange les éléments qui ne sont pas dans la bonne partition. Le partitionnement s'achève lorsque les indices se croisent.

Par exemple, si nous choisissons comme pivot la valeur 350, le partitionnement provoque deux échanges, mis en évidence ci-dessous :



À l'issue du partitionnement, la liste est organisée de la façon suivante :



De plus, les affirmations suivantes sont vérifiées :

$$\forall k \in [1, i-1], \text{clé(ième(l,k))} \leqslant \text{pivot}$$
$$\forall k \in [j+1, \text{longueur(l)}], \text{clé(ième(l,k))} \geqslant \text{pivot}$$
$$\forall k \in [j+1, i-1], \text{clé(ième(l,k))} = \text{pivot}$$

Le partitionnement d'une liste entre les rangs *gauche* et *droit* autour d'un pivot est donné ci-dessous :

```
Algorithme partitionnement(l, gauche, droit)
   {Partitionnement d'une liste l autour d'un pivot}
   {entre les rangs gauche et droit}
   i ← gauche
   j ← droit
   pivot ← {choisir le pivot}
   répéter
      tantque clé(ième(l,i))<clé(pivot) faire i ← i+1 fintantque
      tantque clé(ième(l,j))>clé(pivot) faire j ← j-1 fintantque
      si i⩽j alors
         échanger(l,i,j)
         i ← i+1
         j ← j-1
      finsi
      {∀k∈[1,i-1], clé(ième(l,k))⩽pivot}
      {∀k∈[j+1,longueur(l)], clé(ième(l,k))⩾pivot}
   jusqu'à i>j
   {∀k∈[j+1,i-1], clé(ième(l,k))=pivot}
   {∀k∈[1,i-1], clé(ième(l,k))}⩽pivot}
   {∀k∈[j+1,longueur(l)], clé(ième(l,k))⩾pivot}
```

Vous noterez que le balayage des sous-listes avec des tests d'inégalité stricte peut provoquer des permutations inutiles lorsque la liste contient des clés identiques. On pourrait les remplacer par des tests d'inégalité au sens large, mais dans ce cas le pivot ne jouerait plus son rôle de sentinelle. En effet, imaginons que toutes les clés de la liste soient inférieures ou égales au pivot, le parcours réalisé par la première boucle fera sortir la variable i des bornes de la liste. Si on teste l'égalité, il faut prévoir une autre gestion de sentinelle, ou plus simplement récrire le parcours de la façon suivante :

```
tantque i⩽j et clé(ième(l,i))⩽clé(pivot) faire i←i+1 fintantque
tantque i⩽j et clé(ième(l,j))⩾clé(pivot) faire j←j-1 fintantque
```

Le tri d'une liste entre les rangs gauche et droit se poursuit par le partitionnement des deux

partitions créées, selon la même méthode. La méthode de tri rapide écrite récursivement en JAVA pour une sous-liste comprise entre les rangs `gauche` et `droit` est donnée ci-dessous :

```
private void triRapide(Liste l,Comparable c,int gauche,int droit)
{
  int i=gauche, j=droit;
  Object pivot = ((Élément) l.ième((i+j)/2)).clé();
  do {
     while (c.inférieur(((Élément) l.ième(i)).clé(), pivot)) i++;
     while (c.supérieur(((Élément) l.ième(j)).clé(), pivot)) j--;
     if  (i<=j) {
         l.échanger(i,j);
         i++; j--;
     }
     // invariant :
     // ∀k∈[1,i-1], clé(ième(l,k))⩽pivot
     // ∀k∈[j+1, longueur(l)], clé(ième(l,k))⩾pivot
  } while (i<=j);
  // ∀k∈[j+1,i-1], clé(ième(l,k))=pivot
  // ∀k∈[1,i-1], clé(ième(l,k))⩽pivot
  // ∀k∈[j+1, longueur(l)], clé(ième(l,k))⩾pivot
  if (gauche<j) triRapide(l, c, gauche, j);
  if (droit>i) triRapide(l, c, i, droit);
}
```

Pour trier une liste complète, il suffit de donner les valeurs 1 et *longueur(l)*, respectivement, à *gauche* et *droit* lors du premier appel de la méthode. Le tri rapide s'écrit simplement :

```
public void triHoare(Liste l, Comparable c) {
      triRapide(l, c, 1, l.longueur());
}
```

La figure 22.4 montre les différentes étapes du tri sur la liste de référence. Les carrés indiquent les pivots choisis par la méthode précédente, et les ovales entourent les partitions créées autour du pivot après échanges.

Le choix du pivot conditionne fortement les performances du tri rapide. En effet, la complexité moyenne du nombre de comparaisons dans la phase de partitionnement d'une liste de longueur n est $\mathcal{O}(n)$, puisqu'on compare le pivot aux $n-1$ autres valeurs de la liste. Il y a n ou $n+1$ comparaisons. Si le choix du pivot est tel qu'il divise systématiquement la liste en deux partitions de même taille, c'est-à-dire que le pivot correspond à la médiane, le nombre de comparaisons sera égal à $n \log_2 n$. En revanche, si à chaque étape, le choix du pivot divise la liste en deux partitions de longueur 1 et $n-1$, les performances du tri chutent de façon catastrophique et le nombre de comparaisons est $\mathcal{O}(n^2)$. Dans l'exemple donné, le tri est optimal pour les partitionnements successifs de la partition de droite produite à la première étape ; le pivot est à chaque fois la médiane. Pour la partition de gauche, le choix des pivots 230 et 180 est au contraire le plus mauvais. Dans le cas moyen, on a démontré, sous l'hypothèse de clés différentes et équiprobables, que le nombre de comparaisons est $2n\ln(n) \approx 1{,}38n \log_2 n$; sa complexité reste égale à $\mathcal{O}(n \log_2 n)$.

À chaque étape de partitionnement, le nombre d'échanges est dans le meilleur des cas

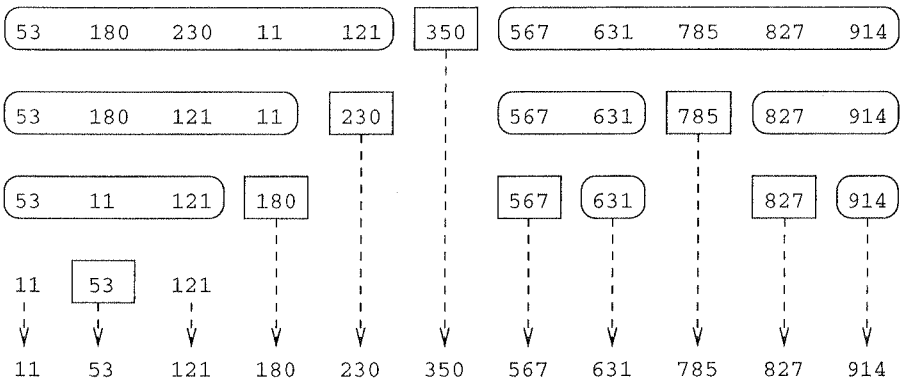


Figure 22.4 – *Tri rapide de la liste de référence.*

égal à 1, dans le pire des cas $\lceil n/2 \rceil$, et dans le cas moyen environ $n/6$. Pour un tri complet, il en résulte que la complexité moyenne du nombre d'échanges est $\mathcal{O}(n \log_2 n)$.

Comment choisir le pivot ? Le choix du pivot doit rester simple, afin de garder toute son efficacité à la méthode. Dans la méthode programmée plus haut, nous avons choisi l'élément du milieu. Nous aurions pu tout aussi bien choisir le pivot au hasard, ou le premier ou le dernier de la liste. Mais attention à ces deux derniers choix, si la liste est déjà triée ou inversement triée, l'algorithme donnera sa plus mauvaise performance. C.A.R. HOARE suggère de prendre la médiane de trois éléments, par exemple le premier, le dernier et celui du milieu.

22.2.5 Comparaisons des méthodes

Nous donnons dans cette section quelques éléments de comparaison des méthodes de tri internes présentées dans ce chapitre, mis en lumière par des résultats expérimentaux obtenus sur un PENTIUM II 266. Les méthodes de tri écrites en JAVA ont été testées pour des listes dont les longueurs varient de 10 à 50 000 et dont les éléments sont tirés au hasard, en ordre croissant et décroissant.

Pour des listes d'environ une centaine d'éléments, toutes les méthodes se valent. Au-delà de cette valeur, il apparaît des différences significatives entre les méthodes simples (sélection directe, tri à bulles, insertion séquentielle et dichotomique) et les méthodes élaborées (tri shell, tri en tas et tri rapide). Il faut environ dix-sept minutes pour trier 25 000 éléments avec une sélection directe, alors qu'une seule seconde suffit au tri rapide.

Parmi les méthodes simples, le tri à bulles a les plus mauvaises performances, sauf dans le cas où la liste déjà est déjà triée, il est alors un peu meilleur que le tri par sélection directe. Les tris par insertion sont meilleurs que les tris par sélection, et dans le cas particulier d'une liste déjà triée, ils donnent les meilleurs résultats de tous les tris puisque leur complexité est égale à $\mathcal{O}(n)$. Dans le cas moyen, le tri par insertion dichotomique n'apporte pas d'amélioration spectaculaire à cause du décalage linéaire. Pour des éléments de grande taille, le décalage est pénalisant et le tri par sélection peut se révéler alors plus performant. En JAVA, toutefois, l'encombrement d'un élément sera toujours celle d'une référence, et les tris par insertion

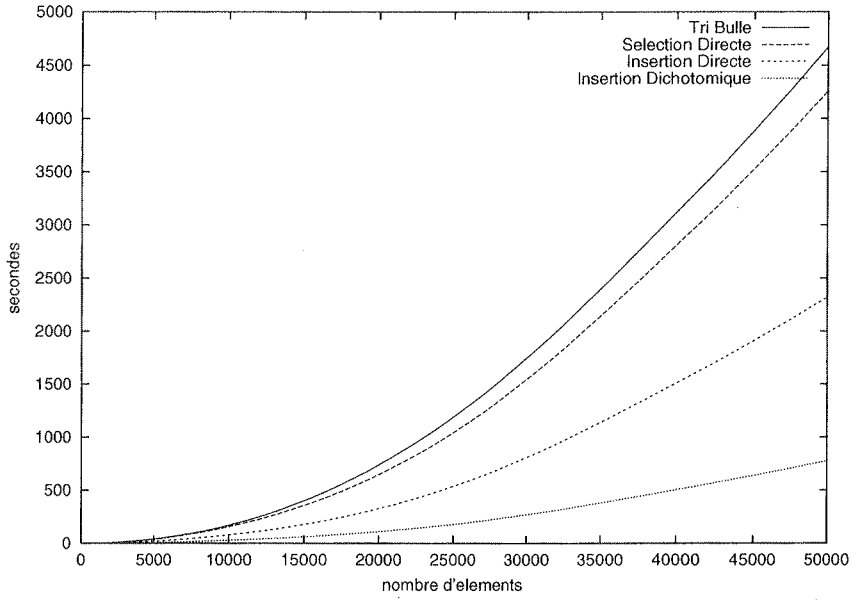


Figure 22.5 – *Méthodes simples.*

resteront meilleurs que ceux par sélection. La figure 22.5 montre des temps d'exécution pour des listes quelconques. Les abscisses donnent la longueur de la liste, et les ordonnées le temps exprimé en secondes.

Parmi les méthodes élaborées, le tri rapide est incontestablement le plus performant. Quelle que soit la façon de choisir le pivot (hasard, milieu ou médiane), les temps de tri sont à peu près identiques. Notez que le tri rapide et le tri en tas possèdent une complexité théorique en nombre de comparaisons assez semblable, et en fait meilleure pour le second. On constate, dans la pratique, que le premier tri est deux fois plus rapide que le second, certainement parce que le nombre de déplacements des éléments est inférieur. D'autre part, certains auteurs ont remarqué qu'à partir d'une certaine taille des partitions, le coût des appels récursifs devient significatif. À partir d'un certain seuil, environ 20 éléments, on substitue au tri rapide une méthode de tri simple, par exemple un tri par insertion séquentielle. Les résultats que nous avons obtenus de façon expérimentale n'ont montré aucune amélioration, mais au contraire une légère dégradation des performances.

Le tri par distances décroissantes donne de meilleurs résultats que le tri en tas lorsque la liste est triée ou triée en ordre inverse. Nous avons vérifié également que la suite $h_{k-1} = 2h_k + 1, h_t = 1$ et $t = \lfloor \log_3 n \rfloor - 1$ offre les meilleurs résultats pour cette méthode.

Les temps d'exécution pour des listes quelconques par les méthodes élaborées sont donnés par les courbes de la figure 22.6.

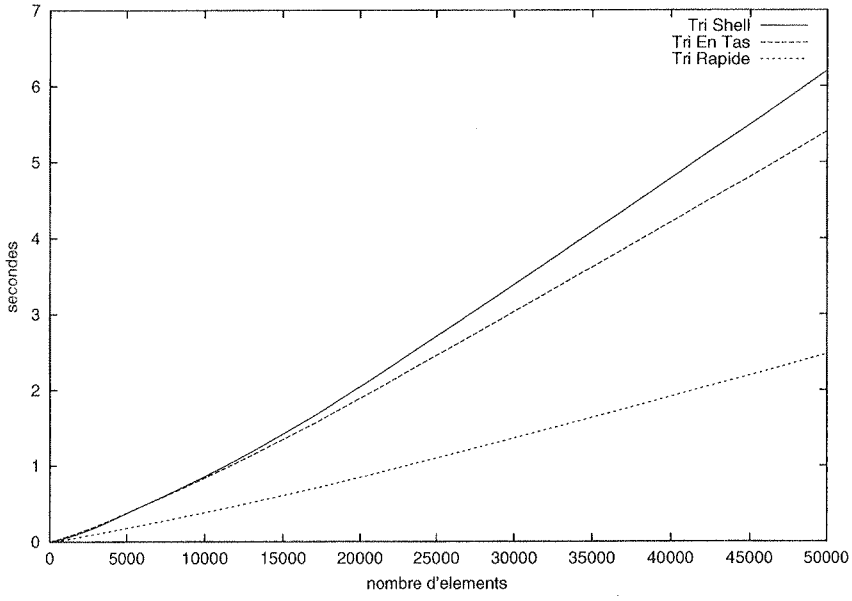


Figure 22.6 – *Méthodes élaborées.*

22.3 TRIS EXTERNES

On utilise des méthodes de tri externes lorsque les données à trier ne peuvent pas toutes être placées en mémoire centrale. C'est une chose qui arrive assez fréquemment dans les applications de gestion et de base de données. Les méthodes de tri externes dépendent des caractéristiques de l'environnement matériel et logiciel. On se place ici dans le cas de la gestion d'éléments placés sur *fichiers séquentiels*. La principale difficulté des tris externes est la gestion des fichiers auxiliaires, et en particulier d'en minimiser leur nombre. Nous présentons dans cette section le tri par *fusion naturelle*, basé sur la distribution de *monotonies* et leur *fusion* par interclassement.

Le tri par fusion naturelle

Soit f le fichier initial contenant les éléments à trier. Ce fichier contient n monotonies naturelles. La méthode de tri nécessite deux fichiers auxiliaires, g et h sur lesquels les n monotonies sont alternativement distribuées. Dans le meilleur des cas, il y aura $n/2$ (ou $n/2$ et $n/2+1$) monotonies sur chacun des fichiers auxiliaires après cette opération de distribution. Le tri consiste ensuite à fusionner deux à deux les monotonies de g et h sur f. Le fichier f contient alors un nombre de monotonies inférieur ou égal à $n/2+1$. Il est clair que la répétition de ce processus fait tendre le nombre de monotonies sur f vers 1 et le fichier est alors trié.

La figure 22.7 montre les trois étapes successives de distribution et de fusion nécessaires au tri de la suite de référence selon cette méthode.

Ce processus de tri fusion naturelle est donné par la méthode JAVA suivante :

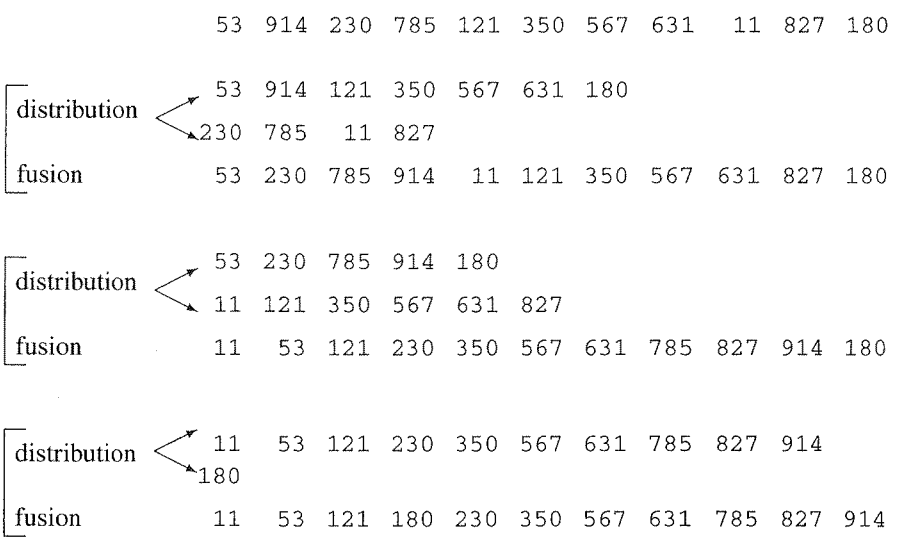


Figure 22.7 – *Étapes du tri par fusion naturelle de la suite de référence.*

```
public void fusionNaturelle(File f, Comparable c)
throws Exception
{
  File g = new File("/tmp/g.dat"),
       h = new File("/tmp/h.dat");
  do { // répartir alternativement les monotonies sur g et h
       distribuer(f,g,h,c);
       // fusionner g et h sur f
  } while (fusionner(g,h,f,c)!=1);
  // f de contient plus qu'une seule monotonie
}
```

Les fichiers d'éléments doivent être considérés comme des fichiers de monotonies pourvues d'opérations spécifiques de manipulation de monotonies. Pour le tri, nous définirons le type abstrait $\mathcal{F}m$ avec les opérations particulières suivantes :

fdf	:	$\mathcal{F}m$	$\rightarrow$	booléen
fdm	:	$\mathcal{F}m$	$\rightarrow$	booléen
copiermonotonie	:	$\mathcal{F}m \times \mathcal{F}m$	$\rightarrow$	$\mathcal{F}m$
copierlesmonotonies	:	$\mathcal{F}m \times \mathcal{F}m$	$\rightarrow$	$\mathcal{F}m \times$ naturel
fusionmonotonie	:	$\mathcal{F}m \times \mathcal{F}m \times$	$\rightarrow$	$\mathcal{F}m$

Les opérations *fdf(f)* et *fdm(f)* indiquent respectivement si la fin du fichier f est atteinte, ou si la fin de sa monotonie courante est atteinte. L'opération *copiermonotonie(f,g)* copie la monotonie courante de f sur g. L'opération *copierlesmonotonies(f,g)* copie sur g toutes les monotonies de f, à partir de la monotonie courante, et retourne le nombre de monotonies copiées. Enfin, l'opération *fusionmonotonie(f,g,h)* écrit sur h la monotonie résultat de la fusion des monotonies courantes de f et g.

La mise en œuvre en JAVA du type abstrait $\mathcal{F}m$ par extension des classes `ObjectInputStream` et `ObjectOutputStream` ne pose pas de difficulté, et elle est d'ailleurs laissée en exercice. Notez que seuls les fichiers d'entrée doivent être traités comme des fichiers de monotonies. Nous appellerons `FichierMonotoniesEntrée` la classe qui les représente.

L'opération de distribution des monotonies sur les deux fichiers auxiliaires est donnée par la méthode `distribuer` :

```
void distribuer(File f1, File f2, File f3, Comparable c)
throws Exception
// Rôle: répartit alternativement les monotonies du fichier f1
// sur les fichiers f2 et f3
{
  FichierMonotoniesEntrée f =
        new FichierMonotoniesEntrée(new FileInputStream(f1),c);
  ObjectOutputStream
        g = new ObjectOutputStream(new FileOutputStream(f2)),
        h = new ObjectOutputStream(new FileOutputStream(f3));
  while (!f.fdf()) {
      f.copierMonotonie(g);
      if  (!f.fdf()) f.copierMonotonie(h);
  }
  f.close(); g.close(); h.close();
}
```

Une fois les monotonies de f réparties sur g et h, il ne reste plus qu'à les fusionner sur f. L'algorithme de fusion est classique, il parcourt simultanément les deux fichiers g et h et fusionne les monotonies deux à deux. La fin de l'un des deux fichiers peut être atteinte avant l'autre. Dans ce cas, il est nécessaire de recopier toutes les monotonies restantes sur f. La méthode retourne le nombre de monotonies écrites sur f.

```
int fusionner(File f1, File f2, File f3, Comparable c)
throws Exception
{
  FichierMonotoniesEntrée
   f = new FichierMonotoniesEntrée(new FileInputStream(f1), c),
   g = new FichierMonotoniesEntrée(new FileInputStream(f2), c);
  ObjectOutputStream
         h = new ObjectOutputStream(new FileOutputStream(f3));
  int nbMono = 0;
  while (!f.fdf() && !g.fdf()) {
      f.fusionMonotonie(g,h);
      nbMono++;
  }
  // f.fdf() ou g.fdf()
  if (!f.fdf())
      // copier toutes les monotonies de f à la fin de h
      nbMono+=f.copierLesMonotonies(h);
  else
      if (!g.fdf())
```

```
            // copier toutes les monotonies de g à la fin de h
            nbMono+=g.copierLesMonotonies(h);
    return nbMono;
}
```

Dans les algorithmes de tri externe, seule la complexité associée aux déplacements des éléments est vraiment intéressante dans la mesure où le temps nécessaire à un accès en mémoire secondaire est d'un ordre de grandeur en général bien supérieur à celui d'une comparaison en mémoire centrale.

À chaque étape *distribution-fusion*, le nombre de monotonies est au pire divisé par deux. Dans le pire des cas (fichier à n éléments trié à l'envers), il y aura $\lceil \log_2 n \rceil$ étapes, chaque étape imposant n déplacements. Le nombre maximal de déplacements est donc $n\lceil \log_2 n \rceil$. Dans le cas moyen, si m est la longueur moyenne des monotonies, le nombre d'étapes sera égal à $\lceil \log_2(n/m) \rceil$.

22.4 EXERCICES

Exercice 22.1. Écrivez la méthode *triIdiot*, qui trie une liste de longueur n dans un temps non borné. La méthode du tri est la suivante : tirez deux indices différents au hasard compris entre 1 et n, échangez les deux éléments associés et regardez si la table est triée. Si la table est triée, on arrête ; sinon on recommence. Testez cette méthode ? Jusqu'à quelle longueur de liste cette méthode donne-t-elle un résultat en un temps raisonnable ?

Exercice 22.2. On dit qu'un tri est *stable* si, pour deux éléments qui possèdent la même clé, l'ordre de leur position initiale est conservé dans la liste triée. Parmi les tris présentés dans ce chapitre, indiquez ceux qui sont stables.

Exercice 22.3. Le *tri par fusion* procède par interclassement de sous-listes triées. L'algorithme de ce tri s'exprime bien récursivement. On divise la liste à trier en deux sous-listes de même taille que l'on trie récursivement par fusion. Les deux sous-listes triées sont ensuite fusionnées par interclassement. Le tri par fusion d'une liste entre les rangs gauche et droit est donné par :

```
Algorithme triFusion(l, gauche, droit)
   si gauche<droit alors
      milieu ← (gauche+droit)/2
      triFusion(l, gauche, milieu)
      triFusion(l, milieu+1, droit)
      fusion(l, gauche, milieu, droit)
   finsi
```

Quelle est la complexité de ce tri ? Quel en est son principal inconvénient ? Programmez en JAVA le tri par fusion.

Exercice 22.4. Programmez le tri par distances décroissantes avec la séquence, proposée par R. SEDGEWICK, 1,5,19,41,109,209,... dont les termes sont calculés de façon croissante

à partir des fonctions $9 \times 4^k - 9 \times 2^k + 1$ et $4^k - 3 \times 2^k + 1$. Cette séquence donne en pratique les meilleurs résultats.

Exercice 22.5. Programmez la méthode du tri rapide et faites des tests comparatifs en choisissant pour pivot: 1) le premier élément de la liste 2) la moyenne de la première et de la dernière valeur de la liste 3) la valeur médiane des trois premiers éléments différents.

Exercice 22.6. Modifiez (légèrement) l'algorithme du tri rapide afin de retourner la valeur médiane d'une liste. On rappelle que la valeur médiane divise une liste en deux sous-listes de même taille telles que tous les éléments de la première sont inférieurs ou égaux à la médiane, et tous éléments de la seconde lui sont supérieurs ou égaux.

Exercice 22.7. Nous avons vu que la complexité théorique la pire d'un tri par comparaison est $\mathcal{O}(n \log_2 n)$. Il est toutefois possible de trier en $\mathcal{O}(n)$ lorsqu'on possède des informations sur les éléments. Prenons, par exemple, une liste de n entiers à trier (ces n entiers sont distincts et compris entre 1 et n. L'algorithme suivant trie la liste `l` dans un tableau `t` sans aucune comparaison et avec n transferts.

```
pourtout i de 1 à n faire
    t[ième(l,i)] ← ième(l,i)
finpour
```

Modifiez cet algorithme de façon à faire un tri sur place (sa complexité doit rester en $\mathcal{O}(n)$).

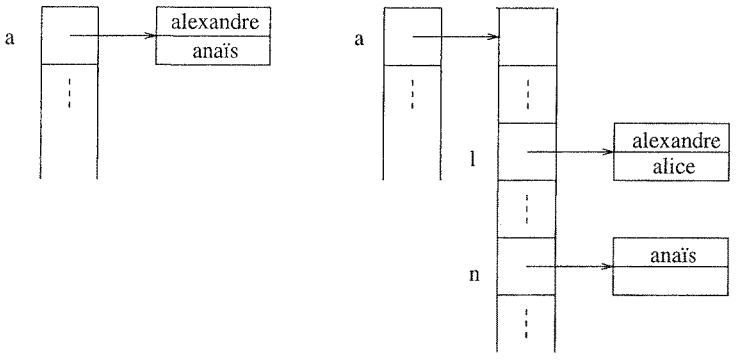
Exercice 22.8. La méthode de tri par paquets généralise la méthode de l'exercice précédent. Pour une suite de n entiers pris sur l'intervalle $[1,m]$, on utilise un tableau `t` de longueur m initialisé à 0. Pour chaque entier i de la suite, on incrémente `t[i]` de un. À la fin, il suffit de parcourir le tableau `t` du début pour obtenir la suite triée. Écrivez l'algorithme qui met en œuvre cette méthode.

Exercice 22.9. Nous voulons trier une suite de chaînes de caractères dans l'ordre lexicographique (*i.e.* celui du dictionnaire). Pour cela, nous allons regrouper les chaînes dans une table structurée par paquets de longueur maximale m. Ces paquets sont constitués par des chaînes de caractères débutant par un même groupe de k caractères, le $k+1^{\text{e}}$ permettant de différencier chaque chaîne. Lorsque un paquet est plein, on crée une table d'indirection permettant de discriminer sur cette $k+1^{\text{e}}$ lettre. Par exemple, les chaînes *alexandre* et *anaïs* sont dans un paquet ($m = 2$) et sont différenciées par leur deuxième lettre ($k = 2$). La figure suivante montre l'effet de l'ajout de la chaîne *alice*.

Définissez la structure de données qui représente la table qui mémorise les chaînes de caractères. Écrivez l'algorithme d'insertion d'une chaîne dans la table, puis l'algorithme qui affiche sur la sortie standard toutes les chaînes de caractères dans l'ordre lexicographique.

Exercice 22.10. Programmez la classe `FichierMonotoniesEntrée` du tri externe par fusion naturelle donné dans ce chapitre.

Exercice 22.11. Le tri externe présenté interclasse les monotonies qui existent *naturellement* dans le fichier à trier. Une autre façon de procéder est de lire les éléments du fichier par paquets de taille m, de trier en mémoire centrale chaque paquet à l'aide d'un tri interne, et de distribuer les monotonies ainsi construites sur les fichiers auxiliaires. Le tri se poursuit par



interclassement comme précédemment. Si on dispose de $2k$ fichiers, une moitié en lecture, et l'autre en écriture. Une opération de fusion consistera à interclasser au plus k monotonies dans un des fichiers en écriture. Quel est le nombre moyen d'étapes nécessaires au tri de n éléments si les monotonies sont de longueur m et s'il y a k fichiers en lecture ? Programmez cette méthode de tri externe pour $k = 4$. Notez que vous pouvez utiliser une file avec priorité pour trouver le minimum des k monotonies lors de l'interclassement.

Chapitre 23

Algorithmes sur les graphes

Ce chapitre présente quatre algorithmes sur les graphes parmi les plus classiques. Les algorithmes de recherche des composantes connexes et de fermeture transitive d'un graphe sont utiles pour tester la connexité de ses sommets. Parmi les problèmes de cheminement, l'algorithme de DIJKSTRA permet de trouver le chemin le plus court entre un sommet particulier et les autres sommets d'un graphe valué. Enfin, nous terminerons avec l'algorithme de tri topologique qui résout des problèmes d'ordonnancement des graphes orientés sans cycle. La programmation en JAVA de ces algorithmes vient enrichir les classes qui implantent les interfaces `Graphe` et `GrapheValué` données au chapitre 18.

23.1 COMPOSANTES CONNEXES

Une manière de vérifier s'il existe un chemin entre deux sommets est de vérifier s'ils appartiennent à la même *composante connexe*. On rappelle qu'un graphe non orienté est composé d'une ou plusieurs composantes connexes. Une composante connexe est formée de sommets tels qu'il existe toujours un chemin qui les relie. Une façon simple de trouver toutes les composantes d'un graphe est d'utiliser des parcours en profondeur ou en largeur. Nous avons vu à la section 18.4 que de tels parcours à partir d'un sommet initial s accédaient aux sommets pour lesquels un chemin depuis s existait.

On appelle *arbre couvrant* d'un graphe connexe non orienté $G = (X,U)$, le graphe partiel $G' = (X,U')$ tel que $U' \subseteq U$ et G' est connexe et sans cycle. Pour trouver les composantes connexes d'un graphe, on construit les arbres couvrants qui leur sont associés lors du parcours complet du graphe.

Le graphe de la figure 23.1 possède deux composantes connexes. Les deux arbres couvrants associés à ce graphe sont donnés par la figure 23.2.

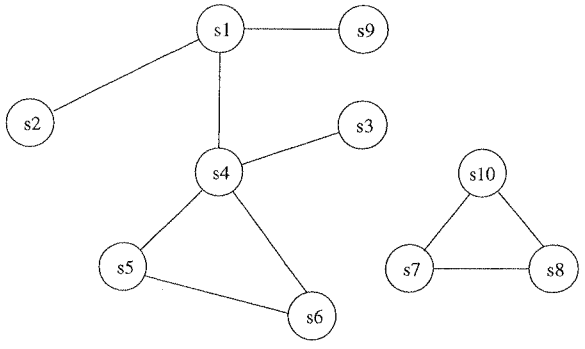


Figure 23.1 – *Graphe avec deux composantes connexes.*

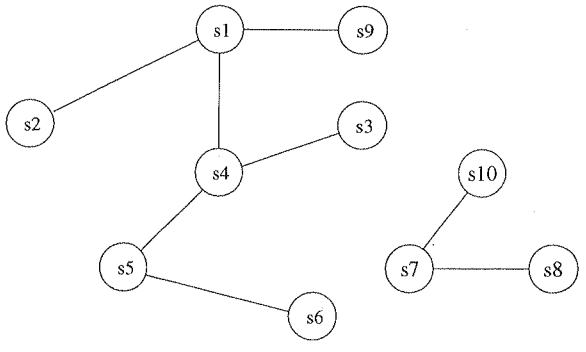


Figure 23.2 – *Deux arbres couvrants issus du graphe de la figure 23.1.*

L'algorithme suivant construit une liste d'arbres couvrants A_i, associés à chaque composante connexe d'un graphe non orienté. Il fait un parcours en profondeur des sommets du graphe. Chaque sommet non marqué s est ajouté dans l'arbre couvrant A_i. Chacun de ses successeurs non marqués x est ajouté dans A_i et une arête (s,x) est créée. La construction de l'arbre couvrant se poursuit récursivement avec les successeurs x.

```
Algorithme Composantes-Connexes(G, L)
   {Construit une liste L des arbres couvrants de chaque
    composante connexe du graphe non orienté G}
   pourtout s de G faire
       si non marqué(s) alors
            {construire la ième composante connexe Ai
             dont la racine est le sommet s}
            ajouter le sommet s à Ai
            cConnexes(G, s, Ai)
            ajouter(L, Ai)
       finsi
   finpour
```

```
Algorithme cConnexes(G, s, A)
   {Parcourir en profondeur des successeurs du sommet s
    et construire la composante connexe A}
   mettre une marque sur le sommet s
   pourtout x de G tel que ∃ une arête(s,x) faire
        si non marqué(x) alors
             {créer une arête (s,x)}
             ajouter le sommet x à A
             ajouter l'arête (s,x)
             cConnexes(G, x, A)
        finsi
   finpour
```

La complexité de la création des arbres couvrants est la même que celle du parcours complet en profondeur d'un graphe, soit $\mathcal{O}(n^2)$ si le graphe est représenté par une matrice d'adjacence, et $\mathcal{O}(\max(n,p))$ si le graphe est représenté par des listes d'adjacence.

Nous allons évoquer quelques problèmes complémentaires liés à la connexité des graphes, mais qui ne seront pas développés. Pour une composante connexe, il est possible d'associer plusieurs arbres couvrants. Il est facile de voir que, si on supprime l'arête $(s5,s6)$, dans l'arbre couvrant à gauche de la figure 23.2 page 320 et que si on crée l'arête $(s4,s6)$, on obtient un nouvel arbre couvrant pour la première composante connexe. Lorsque les graphes sont valués, c'est-à-dire si une valeur est associée à chaque arête, un problème classique est celui de la recherche de l'arbre couvrant *minimal*, c'est-à-dire l'arbre dont la somme des valeurs des arêtes est la plus petite. L'algorithme de KRUSKAL et celui de PRIM apportent tous les deux une solution à ce problème.

Un graphe orienté est *fortement* connexe, si pour tout sommet s et s', il existe à la fois un chemin de s vers s' et un chemin de s' vers s. Pour ces graphes, il s'agit donc de rechercher les composantes fortement connexes les plus grandes. Pour cela, les algorithmes mis en œuvre dérivent du parcours en profondeur.

23.2 FERMETURE TRANSITIVE

Le calcul de la *fermeture transitive* d'un graphe sert aussi à vérifier l'existence, ou non, d'un chemin entre deux sommets d'un graphe. Mais, plutôt que d'avoir à faire une vérification pour deux points particuliers du graphe, il est bien souvent très utile, d'obtenir, au *préalable*, cette connaissance pour tous les sommets du graphe.

La fermeture transitive d'un graphe $G = (X,U)$ est égale au graphe $G^+ = (X,U^+)$ tel que pour tout arc (x,y) de G^+, il existe un chemin de longueur supérieure ou égale à un dans G d'extrémité initiale x et d'extrémité finale y. La fermeture transitive est dite *réflexive*, et notée G^*, si la longueur du chemin peut être égale à 0. Dans ce cas, tout arc (x,x) est toujours présent dans G^*. Remarquez que la notion de fermeture transitive n'est pas propre aux graphes. En fait, elle peut être appliquée à toute relation binaire sur un ensemble $\mathcal{E}$ d'éléments dont il s'agit de connaître l'accessibilité par transitivité.

L'algorithme du calcul de la fermeture transitive d'un graphe G, connu sous le nom d'algorithme de FLOYD–WARSHALL, consiste à ajouter, de façon incrémentale, pour tout sommet s de G, un arc (x,y) s'il existe un chemin entre x et s et un chemin entre s et y. L'algorithme procède de façon itérative en parcourant l'ensemble des sommets du graphe. Il s'agit donc, à chaque fois qu'une relation de transitivité entre deux sommets x et y peut être établie, d'enregistrer la relation entre x et y (voir la figure 23.3 page 322).

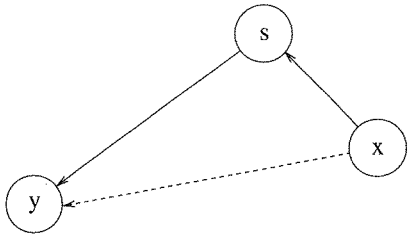


Figure 23.3 – *Relation de transitivité entre les sommets x et y.*

L'algorithme FLOYD–WARSHALL appliqué à un graphe $G = (X,U)$ s'exprime formellement comme suit :

```
Algorithme fermeture-transitive(G)
   pourtout s de X faire
     pourtout x de X faire
        pourtout y de X faire
          si s ≠ x ≠ y et arc(x, s) et arc(s,y) alors
              si ∄ arc(x,y) alors ajouter l'arc (x,y) finsi
          finsi
        finpour
     finpour
   finpour
```

La figure 23.4 montre les arcs qui ont été ajoutés au graphe de la figure 18.1 page 207 par le calcul de cet algorithme, et la figure 23.5 montre la fermeture transitive de ce graphe.

La complexité de cet algorithme est $\mathcal{O}(n^3)$, pour un graphe G à n sommets. Cet algorithme est assez coûteux, et il est important que les opérations de test de l'existence et d'ajout d'un arc dans le graphe soient en $\mathcal{O}(1)$. La représentation d'un graphe par une matrice sera alors préférable.

L'implantation en JAVA

La programmation en JAVA ne pose pas de difficulté particulière. Notez l'utilisation d'énumération pour obtenir tous les sommets du graphe. Les tests d'existence d'un arc avec le sommet courant s permet d'éviter certaines exécutions de boucle, mais l'algorithme n'en demeure pas moins borné par n^3, et reste bien en $\mathcal{O}(n^3)$.

```
public void fermetureTransitive() {
 Énumération e1=grapheÉnumération();
 while (! e1.finÉnumération()) {
    Sommet s=(Sommet) e1.élémentSuivant();
    Énumération e2=grapheÉnumération();
```

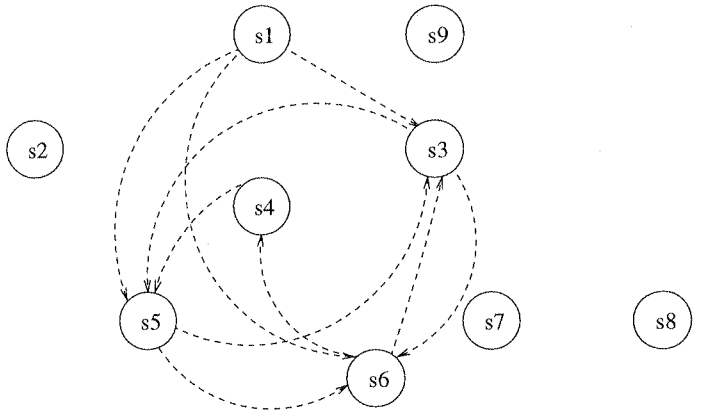


Figure 23.4 – *Arcs issus de la fermeture transitive du graphe de la fig. 18.1.*

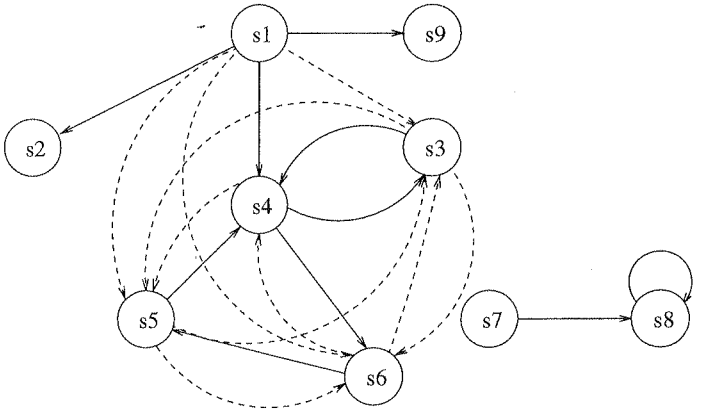


Figure 23.5 – *fermeture transitive du graphe de la figure 18.1.*

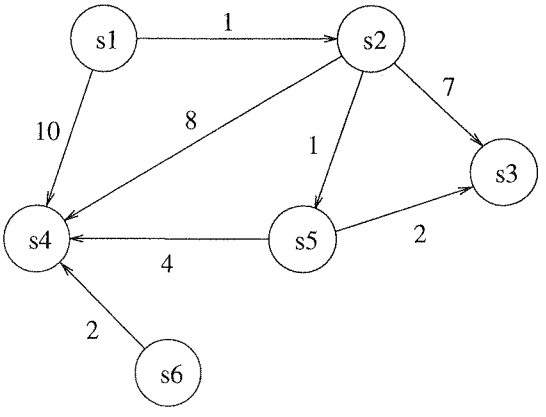
```
    while (! e2.finÉnumération()) {
        Sommet x=(Sommet) e2.élémentSuivant();
       if (x != s && arc(x,s)) {
            Énumération e3=grapheÉnumération();
            while (! e3.finÉnumération()) {
               Sommet y=(Sommet) e3.élémentSuivant();
               if (x != y && y != s && arc(s,y))
                    if (! arc(x,y))
                        // l'arc n'est pas déjà présent ⇒ l'ajouter
                        ajouterArc(x,y);
            }
         }
    }
 }
} // fin fermetureTransitive
```

23.3 PLUS COURT CHEMIN

Nous appellerons *distance*(x,y), la distance entre deux sommets x et y d'un graphe $G = (X,U)$ orienté valué, définie comme la somme des valeurs associées à chaque arc du chemin qui relie x et y. L'objet des algorithmes de recherche de plus court chemin est la recherche de la distance *minimale* entre deux sommets. L'algorithme de DIJKSTRA que nous allons présenter maintenant, permet de calculer la distance minimale entre un sommet *source* et tous les autres sommets d'un graphe valué orienté. Cet algorithme nécessite des valeurs d'arc positives ou nulles.

L'algorithme de DIJKSTRA construit de façon itérative un ensemble solution S formé de couples $x = (s_x,d_x)$, tels que pour tout sommet $s_x \in G$, d_x est égale à la distance minimale entre un sommet source s et le sommet s_x. S'il n'existe pas de chemin pour un sommet s_x, par convention, sa distance avec s est égale à l'infini, noté ∞.

Ainsi, pour le graphe donné ci-dessous et le sommet source $s1$, l'algorithme retourne l'ensemble $S = \{(s1,0),(s2,1),(s3,4),(s4,6),(s5,2),(s6,\infty)\}$.



Initialement, l'ensemble solution S est vide, et un ensemble E est formé de couples (s_x,d_x), tels que :

$$d_x = \begin{cases} 0 & \text{si } s_x = s \\ \infty & \text{si } s_x \neq s \end{cases}$$

Pour ce graphe, les valeurs initiales des ensembles S et E sont telles que :

$S = \emptyset$
$E = \{(s1,0),(s2,\infty),(s3,\infty),(s4,\infty),(s5,\infty),(s6,\infty)\}$

L'ensemble S est construit progressivement de façon itérative. À chaque itération, il existe un élément $m = (s_m,d_m) \in E$ de distance minimale, telle que $\forall x \in E$, $d_m = min(d_x)$. Cet élément est retiré de l'ensemble E et ajouté dans S. *Affirmation* : la distance d_m est la distance du plus court chemin entre s et s_m. Ensuite, les distances d_x de chaque $x \in E$ qui possède un arc avec s_m sont recalculées de telle façon que :

```
si d_m + valeurArc(s_m, s_x) < d_x alors
    d_x ← d_m + valeurArc(s_m, s_x)
finsi
```

où valeurArc(s_m,s_x) est la valeur de l'arc entre le sommet s_m et le sommet s_x. À la dernière itération, l'ensemble E est vide, et S contient la solution.

L'algorithme appliqué au graphe de la page précédente produit les six itérations suivantes :

S	E
{($s1$,0)}	{($s2$,1),($s3$,∞),($s4$,10),($s5$,∞),($s6$,∞)}
{($s1$,0),($s2$,1)}	{($s3$,8),($s4$,10),($s5$,2),($s6$,∞)}
{($s1$,0),($s2$,1),($s5$,2)}	{($s3$,4),($s4$,6),($s6$,∞)}
{($s1$,0),($s2$,1),($s5$,2),($s3$,4)}	{($s4$,6),($s6$,∞)}
{($s1$,0),($s2$,1),($s5$,2),($s3$,4),($s4$,6)}	{($s6$,∞)}
{($s1$,0),($s2$,1),($s5$,2),($s3$,4),($s4$,6),($s6$,∞)}	{}

L'algorithme du plus court chemin de DIJKSTRA s'écrit formellement comme suit :

Algorithme Plus-Court-Chemin(G, s)

```
{initialisations}
S ← ∅
E ← {(s_x,d_x) / ∀s_x ∈ G, d_x = 0 si s_x = s et d_x = ∞ si s_x ≠ s}
{construire l'ensemble S des plus courts chemins}
tantque E ≠ ∅ faire
    {Invariant : soit m ∈ E, ∀k∈E, d_m = min(d_k)}
    {            ∀k∈S, d_k = distance minimale entre s et s_k}
    E ← E - {m}
    S ← S ∪ {m}
    {recalculer les distances dans E}
    pourtout x de E tel que ∃ un arc(s_m, s_x) faire
        si d_m + valeurArc(s_m, s_x) < d_x alors
            d_x ← d_m + valeurArc(s_m, s_x)
        finsi
    finpour
fintantque
{S = { (s_x,d_x), ∀s_x ∈G, d_x = distance minimale entre s et s_x} }
rendre S
```

Montrons que cet algorithme calcule bien l'ensemble solution S des plus courts chemins, tel que $\forall x \in S$, d_x est le plus court chemin de s au sommet s_x. Commençons par le choix de m. Il faut montrer que d_m est bien la distance du plus court chemin de s à s_m. Si ce n'était pas le cas, il existerait un $n \in E$ (voir la figure 23.6), tel que d_n + *distance*(s_n,s_m) $\leqslant d_m$. Or, on a $d_m \leqslant d_n$ et, par hypothèse, les valeurs des arcs ne peuvent être négatives. Donc n ne peut exister, et m possède le sommet s_m de distance minimale par rapport à s, et il est ajouté dans S.

On en déduit par récurrence, que tous les prédécesseurs de s_m, dans le plus court chemin de s à s_m, appartiennent exclusivement à S. De même, pour tout $x = (s_x,d_x) \in E$, avec $d_x \neq \infty$, d_x est la distance d'un *meilleur* chemin entre s et s_x, à l'itération courante, et dont

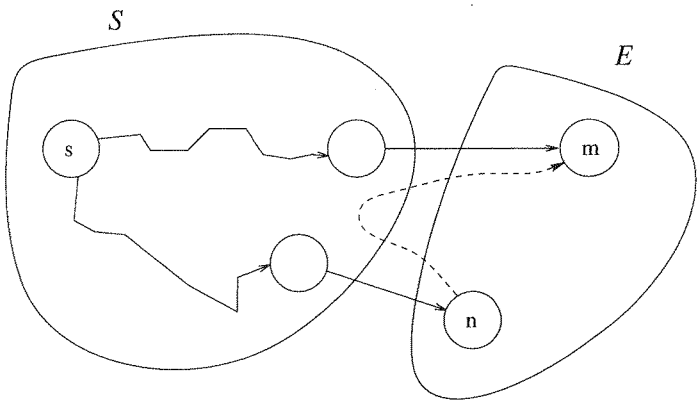


Figure 23.6 – *La distance entre s et m est minimale.*

les sommets prédécesseurs de s_x sont dans S. Il est évident, que la modification des valeurs des distances dans E, après l'ajout de m dans S, maintient cette dernière propriété, et permet, de trouver un meilleur chemin, s'il existe, passant obligatoirement par s_m.

23.3.1 L'implantation en Java

Dans cette section, nous présentons l'écriture en JAVA de la méthode `plusCourtChemin` selon l'algorithme de DIJKSTRA. Cette méthode est ajoutée à la classe `GrapheValué`. Les couples des ensembles E et S sont représentés par le type `Élément` donné à la page 241. La valeur de l'élément sera le sommet, et la clé sa distance au sommet source. Pour accroître la lisibilité de la méthode `plusCourtChemin`, nous définissons les trois méthodes privées suivantes :

```
// Cette méthode retourne le sommet de l'élément e
private Sommet sommet(Élément e);
// Cette méthode retourne la distance de l'élément e
private int dist(Élément e);
// Cette méthode change la distance de l'élément e
private void changerDist(Élément x, int d);
```

L'écriture de ces méthodes ne pose aucune difficulté et sont laissées en exercice. Notez que pour la dernière méthode, il faut ajouter à la classe `Élément` une méthode `changerClé`, pour modifier la valeur de la clé d'un élément.

Les ensembles E et S sont des objets de type `Ensemble`, une interface qui définit les opérations d'un type abstrait *Ensemble*, dont l'implantation sera discutée à la section suivante. Pour le calcul des plus courts chemins, la classe `Ensemble` devra au moins définir les méthodes suivantes :

```
// Cette méthode ajoute l'élément e à l'ensemble courant
public void ajouter(Object e);
```

```
// Cette méthode retourne true si l'ensemble courant est vide,
// et false sinon
public boolean vide();
// Cette méthode supprime le plus petit élément de l'ensemble courant
// et retourne sa valeur
public Object supprimerMin();
// Cette méthode retourne une énumération des éléments de l'ensemble
public Énumération ensembleÉnumération();
```

Nous pouvons donner l'écriture complète de la méthode qui retourne l'ensemble des plus courts chemins d'un sommet s à tous les autres sommets du graphe.

```
// calcul des plus courts chemins entre le sommet s
// et les autres sommets du graphe, selon la méthode de DIJKSTRA
public Ensemble plusCourtChemin(Sommet s) {
    final int INFINI=Integer.MAX_VALUE;
    Ensemble E = new EnsembleListe(),
             S = new EnsembleListe();
    // initialiser l'ensemble E
    Énumération g=grapheÉnumération();
    E.ajouter(new Élément(s, new Integer(0)));
    while (!g.finÉnumération()) {
        Sommet x=(Sommet) g.élémentSuivant();
        if (x != s)
           E.ajouter(new Élément(x, new Integer(INFINI)));
    }
    // construire l'ensemble S des plus courts chemins
    while (!E.vide()) {
        // soit m=(s_m,d_m)∈E, ∀k∈E, s_m = min(d_k)
        // ∀k∈S, d_k = distance minimale entre s et s_k
        Élément m=(Élément) E.supprimerMin();
        S.ajouter(m);
        // pour tout sommet x de E qui possède un arc avec m
        Énumération e=E.ensembleÉnumération();
        while (!e.finÉnumération()) {
            Élément x=(Élément) e.élémentSuivant();
            if (arc(sommet(m),sommet(x))) {
               int d=dist(m) + valeurArc(sommet(m),sommet(x));
               if (d<dist(x))
                     changerDist(x,d);
            }
        }
    }
    // S = {(s_x,d_x), ∀s_x ∈ this, d_x = distance minimale entre s et x}
    return S;
}
```

23.3.2 Complexité de l'algorithme

La complexité de l'algorithme de DIJKSTRA dépend de la représentation de l'ensemble E. Nous allons nous intéresser au nombre de comparaisons effectuées dans le pire des cas par l'algorithme pour un graphe de n sommets et p arcs.

Le corps de la boucle principale est effectué n fois. Une première série de comparaisons est faite lors de la recherche de l'élément m dans l'ensemble E. Si cet ensemble est représenté par une liste linéaire non ordonnée, le nombre de comparaisons pour rechercher et supprimer m, est égal au cardinal de E. Il y aura donc en tout $n(n+1)/2$ comparaisons, la complexité est alors $\mathcal{O}(n^2)$. Cette complexité peut être améliorée : si l'ensemble E est représenté par un tas, l'accès à m est en $\mathcal{O}(1)$, et sa suppression demande au pire $\log_2(n)$, soit au total $n \log_2(n)$ comparaisons. Dans la seconde boucle, le test de vérification d'adjacence des sommets s_m et s_x est exécuté $n(n-1)/2$ fois. La complexité est $\mathcal{O}(n^2)$. Mais, on peut remarquer que le test d'ajustement des distances n'est à faire que $d^+(s_m)$ fois, le demi-degré extérieur du sommet de s_m. Si le nombre d'arcs p est petit devant son maximum, n^2, et si l'accès au ième successeur du sommet s_m peut être exécuté en $\mathcal{O}(1)$, il sera plus judicieux de récrire la seconde boucle de la façon suivante :

```
pourtout i de 1 à degréExt(s_m) faire
    s_x ← ièmeSucc(s_m, i) dans E
    si d_m + valeurArc(s_m, s_x) < d_x alors
        d_x ← d_m + valeurArc(s_m, s_x)
    finsi
finpour
```

Toutefois, cette écriture nécessite un moyen de retrouver le d_x correspondant au s_x pour la mise à jour des distances dans E. Si cette correspondance peut être obtenue en $\mathcal{O}(1)$, la complexité de la modification des distances est $\mathcal{O}(p)$.

Au total, la complexité de l'algorithme de DIJKSTRA est $\mathcal{O}(n^2)$ pour des graphes denses avec E représenté par une liste, alors qu'elle est $\mathcal{O}(n \log_2 n + \max(n^2, p))$ avec E représenté par un tas.

23.4 TRI TOPOLOGIQUE

Le tri topologique définit un ordre (partiel) sur les sommets d'un graphe orienté sans cycle. Une relation *apparaît-avant* compare deux sommets du graphe. Le tri retourne comme résultat une liste linéaire de sommets ordonnés de telle façon qu'*aucun* sommet n'apparaît *avant* un de ses prédécesseurs. Cette relation impose de fait que le graphe soit sans cycle.

La relation d'ordre du tri topologique est partielle, et le résultat du tri peut alors ne pas être unique, dans la mesure où deux sommets peuvent ne pas être comparables. Prenons l'exemple trivial de la figure 23.7. Le tri de ces trois sommets rend les suites $< s1\ s2\ s3 >$ ou $< s2\ s1\ s3 >$.

La liste de sommets L est construite de façon itérative par l'algorithme du tri topologique que nous allons décrire maintenant. Initialement, on associe, dans une table *nbpred*, à chaque sommet de G son nombre de prédécesseurs, c'est-à-dire son demi-degré intérieur. Tous les sommets sans prédécesseur sont mis dans un ensemble E. Puisque le graphe est sans cycle,

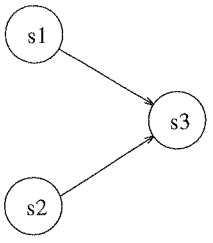


Figure 23.7 – *Les sommets s1 et s2 sont incomparables.*

il existe au moins un sommet sans prédécesseur, et $E \neq \emptyset$. Un sommet s de E est choisi, supprimé de E et ajouté à L. Le nombre de prédécesseurs de tous les sommets successeurs de s, qui appartiennent à G mais pas à L, est alors décrémenté de 1. Si après décrémentation, le nombre de prédécesseurs d'un sommet x est égal à 0, alors x est ajouté à E. Lorsque E est vide, tous les sommets de G sont dans L, et le tri est achevé. L'algorithme repose sur l'affirmation que les prédécesseurs de tous les sommets de E sont dans L. Notez que le choix du sommet s dans E est quelconque, puisque les sommets de E ne sont pas comparables. De façon formelle, l'algorithme du tri topologique s'écrit :

```
Algorithme Tri-Topologique(G, L)
   {Antécédent : G graphe orienté sans cycle}
   {Conséquent : L liste des sommets de G ordonnés}

   {construire l'ensemble E et la table des prédécesseurs}
   E ← ∅
   pourtout x de G faire
       nbpred[x] ← d⁻(x)
       si nbpred[x]=0 alors E ← E∪{x} finsi
   finpour
   {E contient au moins un sommet}
   tantque E ≠ ∅ faire
      {Invariant : ∀x∈E, nbpred[x]=0 et arc(y,x) ⇒ y∈L}
      soit s∈E
        ajouter(L,s)
        E ← E-{s}
        pourtout x de G adjacent à s faire
           {Invariant : x∉L}
           nbpred[x] ← nbpred[x]-1
           si nbpred[x]=0 alors E ← E∪{x} finsi
        finpour
   fintantque
   {L contient tous les sommets de G ordonnés}
   rendre L
```

Cet algorithme appliqué au graphe donné par la figure 23.8 de la page 330 retourne la liste $< s1\ s4\ s3\ s2 >$. Le tableau suivant montre les valeurs successives prises par E et L, ainsi que le nombre de prédécesseurs des sommets de G qui ne sont pas encore dans L.

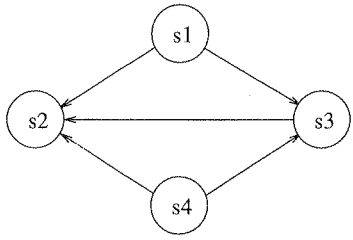


Figure 23.8 – *Tri topologique.*

E	L	nbpred
$s1$,$s4$	$\emptyset$	nbpred[$s2$] = 3, nbpred[$s3$]=2
$s4$	$s1$	nbpred[$s2$] = 2, nbpred[$s3$]=1
$s3$	$s1$,$s4$	nbpred[$s2$] = 1
$s2$	$s1$,$s4$,$s3$	
$\emptyset$	$s1$,$s4$,$s3$,$s2$	

23.4.1 L'implantation en Java

La méthode `triTopologique` suivante est ajoutée à la classe `Graphe`, et s'applique au graphe courant **`this`**. L'ensemble E et la liste L sont simplement représentés par deux files dont les éléments sont des objets de type `Sommet`. La table des prédécesseurs est une table d'adressage dispersé dont les valeurs sont de type `Integer` avec une clé d'accès de type `Sommet`. Pour tout sommet, la méthode `sommetAdjacentÉnumération` retourne l'énumération de ses successeurs.

```
public File triTopologique() {
    File E = new FileChaînée(Sommet.class);
    File L = new FileChaînée(Sommet.class);
    Table nbPred=null;
    try {
        nbPred = new HashCodeFermé(new CléSommet());
    }
    catch (ClassNotFoundException e) { }
    // construire l'ensemble E et la table des prédécesseurs
    Énumération g=grapheÉnumération();
    while (!g.finÉnumération()) {
        Sommet s=(Sommet) g.élémentSuivant();
        int np=demiDegréInt(s);
        if (np==0) E.enfiler(s);
        nbPred.ajouter(new Élément(new Integer(np), s));
    }
    // E contient au moins un sommet
    while (!E.estVide()) {
       // Invariant: ∀x∈E, nbPred[x]=0 et arc(y,x)⇒y∈L
        Sommet s=(Sommet) E.premier();
```

```
            E.défiler();
            L.enfiler(s);
            Énumération e=sommetAdjacentÉnumération(s);
            while (!e.finÉnumération()) {
                Sommet x=(Sommet) e.élémentSuivant();
                // Invariant: x∉L
                Élément ex=(Élément) nbPred.rechercher(x);
                int np=((Integer) ex.valeur()).intValue()-1;
                ex.changerValeur(new Integer(np));
                if (np==0) E.enfiler(x);
            }
        }
        // L contient tous les sommets de this ordonnés
        return L;
    }
```

La complexité du tri topologique d'un graphe à n sommets et p arcs est au pire $\mathcal{O}(n + p)$ si le graphe est représenté par des listes d'adjacence et $\mathcal{O}(n^2)$ s'il utilise une matrice d'adjacence.

Si le graphe est représenté par des listes d'adjacence, la création de la table des prédécesseurs demande un parcours des n sommets du graphe, et le calcul du demi-degré de chaque sommet réclame p tests au pire. La complexité est $\mathcal{O}(n \times p)$. Notez que cette complexité peut être ramenée à $\mathcal{O}(n + p)$, si la table est créée lors d'un parcours en profondeur ou en largeur du graphe. Cette amélioration est laissée en exercice. Si le graphe est représenté par une matrice, la complexité de la création de la table est toujours au pire $\mathcal{O}(n^2)$.

Puisqu'il n'a pas de cycle, chaque sommet s du graphe est traité une seule fois dans la phase de tri proprement dite. Le parcours de ses successeurs est alors proportionnel à p. Il en résulte une complexité égale à $\mathcal{O}(n + p)$.

23.4.2 Existence de cycle dans un graphe

Le tri topologique s'applique à un graphe sans cycle. Mais que se passe-t-il si l'algorithme est appliqué à un graphe qui possède un ou plusieurs cycles ? Une telle situation conduit au fait qu'il n'existe pas de sommet s tel que *nbpred[s]=0*. Il en résulte que E est vide et l'algorithme s'arrête. La liste retournée par le tri exclut les sommets qui forment le cycle. Le tri topologique est alors un moyen de vérifier l'absence ou la présence de cycle dans un graphe.

23.4.3 Tri topologique inverse

Une autre méthode pour effectuer un tri topologique est de réaliser un parcours en profondeur postfixe du graphe (voir l'algorithme à la page 216). Chaque sommet s est ajouté dans la liste L *après* le parcours de ses successeurs. Cette méthode est appelée *tri topologique inverse* car la liste obtenue est en ordre inverse, puisque les sommets apparaissent après leurs successeurs.

Pour le graphe de la figure 23.8, cet algorithme construit la liste $< s2\ s3\ s4\ s1 >$, et propose un second tri topologique valide du graphe, $< s1\ s4\ s3\ s2 >$.

La complexité de cette méthode est celle du parcours en profondeur, c'est-à-dire $\mathcal{O}(n^2)$ si le graphe est représenté par une matrice d'adjacence et $\mathcal{O}(\max(n,p))$ s'il est représenté par des listes d'adjacence.

23.4.4 L'implantation en Java

La programmation de la méthode `triTopologiqueInverse` consiste simplement à appeler la méthode `parcoursProfondeurPostfixe` avec une opération qui construit la liste des sommets. Cette opération implante l'interface `Opération`, donnée à la page 218, à laquelle nous avons ajouté la méthode `résultat` qui retourne un résultat final de parcours.

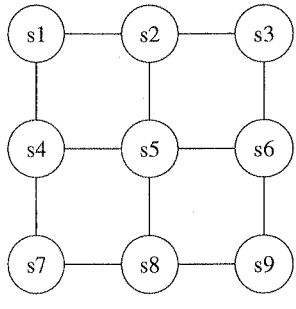
```
public class OpérationTriTopo implements Opération {
    private File f;
    public OpérationTriTopo() {
        f = new FileChaînée(Sommet.class);
    }
    public Object exécuter(Object e) {
        f.enfiler(e);
        return null;
    }
    public Object résultat() {
        return f;
    }
}
```

La méthode `triTopologiqueInverse` s'écrit simplement :

```
public File triTopologiqueInverse() {
    Opération op = new OpérationTriTopo();
    parcoursProfondeurPostfixe(op);
    return (File) op.résultat();
}
```

23.5 EXERCICES

Exercice 23.1. Trouvez l'arbre couvrant du graphe donné par la figure suivante :



Exercice 23.2. Montrez qu'il existe un seul arbre couvrant minimal pour un graphe valué connexe (non orienté) dont les valeurs des arêtes sont positives et distinctes.

Exercice 23.3. Montrez sur un exemple que l'algorithme de DIJKSTRA donne un résultat faux si un arc possède une valeur négative.

Exercice 23.4. Programmez l'algorithme de DIJKSTRA en utilisant un tas et une boucle qui ne parcourt que les successeurs de m dans E.

Exercice 23.5. Pour rechercher le plus court chemin entre tout couple de sommets d'un graphe, il est bien sûr possible d'appliquer itérativement l'algorithme de DIJKSTRA en prenant chacun des sommets du graphe comme source. L'algorithme de FLOYD donne une solution très simple à ce problème. La méthode consiste à considérer chacun des n sommets d'un graphe, appelons-le s, comme intervenant possible dans la chaîne qui lie tout couple de sommets (x,y). Si la distance $d(x,s) + d(s,y)$ est inférieure à $d(x,y)$ alors on a trouvé une distance minimale entre x et y qui passe par le sommet s. L'algorithme de FLOYD utilise une matrice carrée $n \times n$ pour mémoriser les distances calculées au fur et à mesure. Il s'exprime comme suit :

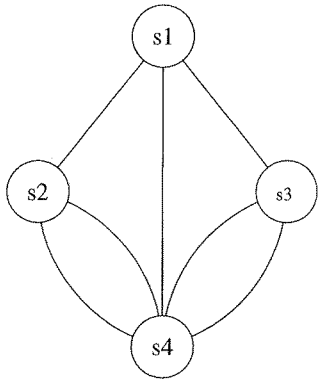
```
Algorithme Floyd(G,d)
   {initialiser la table des distances d}
   ∀x,y∈G, d[numéro(x),numéro(y)] ← valeurArc(x,y)
   {calculer tous les plus courts chemins}
   pourtout s de 1 à n faire
      pourtout x de 1 à n faire
           pourtout y de 1 à n faire
                si d[x,s]+d[s,y] < d[x,y]  alors
                      d[x,y] ← d[x,s]+d[s,y]
                finsi
            finpour
      finpour
   finpour
   rendre d
```

Appliquez cet algorithme sur le graphe de la page 324. Montrez que cet algorithme est valide. Quelle est sa complexité ? Est-il plus efficace que celui qui consiste à appliquer n fois l'algorithme de DIJKSTRA ? Est-ce que l'algorithme de FLOYD précdent peut s'appliquer à des valeurs d'arcs négatives ? Programmez cet algorithme en JAVA.

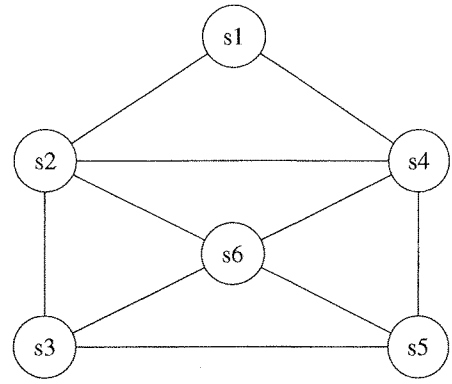
Exercice 23.6. En utilisant une matrice d de booléens (*i.e.* une matrice d'adjacence), modifiez l'algorithme de FLOYD afin de calculer la fermeture transitive réflexive du graphe. Cet algorithme est connu sous le nom d'algorithme de WARSHALL.

Exercice 23.7. Modifiez l'algorithme du tri topologique afin de retourner la valeur booléenne *vrai* si le graphe possède un cycle, et la valeur *faux* dans le cas contraire.

Exercice 23.8. Est-il possible de passer par tous les sommets d'un graphe sans emprunter deux fois la même arête ? Essayez de trouver ce chemin sur les deux graphes donnés par la figure suivante :



(a)



(b)

Ce problème est connu sous le nom de chemin eulérien. EULER [1] a démontré la condition nécessaire à l'existence d'un tel chemin dans un graphe connexe : il doit exister aucun ou deux sommets de degré impair. Si tous les sommets du graphe possèdent un degré pair, il s'agit alors d'un cycle eulérien (*i.e.* les deux extrémités du chemin sont identiques). S'il existe deux sommets de degré impair, ce sont les extrémités du chemin. Appliquez cette condition aux deux graphes précédents Programmez une méthode qui vérifie cette condition.

Pour trouver le chemin eulérien E d'un graphe $G = (X,U)$, on part d'un sommet source (on prendra n'importe quel sommet, si tous les sommets ont un degré pair, ou de l'un des deux sommets de degré impair), et l'on procède selon un parcours en profondeur de la forme :

```
Algorithme parcoursEuler(s, U, E)
   {Antécédent : s sommet origine du parcours
                 U ensemble des arêtes du graphe à parcourir
    Conséquent : E ensemble des arêtes parcourues et U=U-E}

   si ∃v∈X tel que ∃ arête(s,v) ∈U alors
          parcoursEuler(v, U-[s,v], E∪[s,v])
   finsi
```

Au départ, l'ensemble E des arêtes qui forment le chemin eulérien est initialisé à $\emptyset$. Lorsque cet algorithme s'achève deux cas de figure se présentent : soit $U = \emptyset$ et le chemin eulérien de G a été trouvé, soit $U \neq \emptyset$ et seule une partie du graphe a été parcourue. Dans ce dernier cas, il faut recommencer l'algorithme à partir d'un des sommets du chemin E qui possède une arête dans U non parcourue.

Montrez que s'il reste des arêtes non parcourues, le graphe partiel comporte toujours un chemin eulérien. Écrivez en JAVA une méthode qui retourne, s'il existe, le chemin eulérien d'un graphe.

1. L. EULER, mathématicien Suisse (1707-1783). Il a souvent été dit que la résolution de ce problème marque l'origine de la théorie des graphes.

Chapitre **24**

Algorithmes de rétro-parcours

Les problèmes qui mettent en jeu des algorithmes de rétro-parcours sont des problèmes pour lesquels l'algorithme solution ne suit pas une règle fixe. La résolution de ces problèmes se fait par étapes et essais successifs. À chaque étape, plusieurs possibilités sont offertes. On en choisit une et l'on passe à l'étape suivante. Ces choix successifs peuvent conduire à une solution ou à une impasse. Dans ce dernier cas, il faudra revenir sur ses pas et essayer de nouvelles possibilités, éventuellement jusqu'à leur épuisement. C'est une démarche que l'on adopte, par exemple, dans le parcours d'un labyrinthe.

Toutes les étapes, ainsi que les choix que l'on peut faire à chacune de ces étapes, modélisent un arbre de décisions. Chaque nœud de cet arbre est une des étapes où sont proposés les choix. La recherche d'une solution consiste donc à parcourir cet arbre. Les branches qui s'étendent de la racine aux feuilles terminales de l'arbre sont les solutions potentielles. En général, les méthodes de rétro-parcours ne construisent pas explicitement cet arbre de décisions, il est purement virtuel.

Dans ce chapitre, nous présenterons les écritures récursives et itératives des algorithmes de rétro-parcours donnant, si elles existent, une solution particulière ou toutes les solutions possibles. Nous utiliserons ces algorithmes pour résoudre le problème des huit reines, et celui des sous-suites. Enfin, nous terminerons par une application aux jeux de stratégie à deux joueurs.

24.1 ÉCRITURE RÉCURSIVE

L'écriture récursive de l'algorithme de rétro-parcours est basée sur une procédure *d'essai* qui tente d'étendre une solution partielle correcte à l'étape i. À chaque tentative d'extension de la solution partielle, un nouveau candidat est choisi dans une liste de candidats potentiels.

La récursivité permet les retours en arrière lors du parcours de l'arbre. Lorsque la solution partielle est invalide, et l'ensemble des possibilités à l'étape i est épuisé, l'achèvement de la procédure permet de revenir à l'étape précédente. Cette première version recherche une solution particulière, la première.

```
Algorithme essayer(i, correcte)
   {Antécédent : la solution partielle jusqu'à l'étape i-1
                 est correcte}
   {Conséquent : correcte = solution partielle à l'étape i
                            valide ou pas}
   {Rôle : essaye d'étendre la solution au ième coup}
   {initialisation de la liste des possibilités}
   k ← {candidat initial}
   répéter
      {prendre le prochain candidat dans la liste des possibilités}
      k ← {candidat suivant}
      {vérifier si la solution partielle à l'étape i est correcte}
      vérifier(i,ok)
      si ok alors
         enregistrer(i,k)
         si non fini(i) alors
            essayer(i+1,correcte)
            si non correcte alors
               {le choix k à la ième étape conduit à une impasse}
               annuler(i,k)
            finsi
         sinon {on a trouvé une solution}
               correcte ← vrai
         finsi
      finsi
   jusqu'à correcte ou plus de candidats
```

Dans cet algorithme, la fonction *fini* teste si la dernière étape est atteinte ou pas, la fonction *vérifier* teste la validité de la solution partielle à l'étape i, la procédure *enregistrer* mémorise dans la solution partielle le candidat k à l'étape i, et la procédure *annuler* efface de la solution partielle le candidat k à l'étape i.

Pour obtenir toutes les solutions du problème, il suffit, à chaque étape, d'essayer *tous* les candidats possibles. Remarquez que cela correspond au parcours complet de l'arbre de décisions.

```
Algorithme essayer(i)
   {Antécédent : la solution partielle jusqu'à l'étape i-1
                 est correcte}
   {Rôle : essayer d'étendre la solution au ième coup}
   {initialisation de la liste des possibilités}
   pourtout k de la liste des candidats faire
        {prendre le k^e candidat dans la liste des possibilités
         vérifier si la solution partielle à l'étape i est correcte}
        vérifier(i,ok)
```

```
            si ok alors
               enregistrer(i,k)
               si non fini(i) alors essayer(i+1)
                  sinon {on a trouvé une solution} écriresolution
               finsi
               annuler(i,k)
            finsi
         finpour
```

24.2 LE PROBLÈME DES HUIT REINES

Ce problème proposé par C. F. GAUSS en 1850 consiste à placer huit reines sur un échiquier sans qu'elles puissent se mettre en échec mutuellement (selon les règles de déplacement des reines). Il n'y a pas d'algorithme direct donnant une solution et un algorithme de rétro-parcours doit être utilisé.

Nous connaissons l'algorithme, nous allons nous intéresser aux structures de données. Il est évident qu'on ne pourra placer qu'une reine par colonne. Pour chaque colonne c, le choix se réduit donc à la ligne l sur laquelle poser la reine. À partir de ce qui vient d'être dit, il est inutile de représenter l'échiquier par une matrice 8×8, un tableau de huit positions suffit pour représenter une solution :

```
solution type tableau [ [1,8] ] de [1,8]
```

L'algorithme va placer une reine par colonne. Vérifier si une reine est correctement placée nécessite de vérifier s'il n'y a pas de conflit sur la ligne et sur les deux diagonales. Cette vérification doit rester simple. Nous utiliserons trois tableaux de booléens, `ligne`, `diag1` et `diag2` tels que pour une ligne l et une colonne c :

- `ligne[l]` indique si la ligne l est libre ou non ;
- `diag1[k]` indique si la k^e diagonale $\swarrow$ est libre ou non ;
- `diag2[k]` indique si la k^e diagonale $\searrow$ est libre ou non.

La figure 24.1 montre bien à quoi correspondent `diag1` et `diag2` et comment représenter k en fonction de l et c.

On voit donc qu'à partir de la position (l,c), la diagonale $\swarrow$ correspond à l'indice $l + c$ et la diagonale $\searrow$ à l'indice $l - c$. Puisque l et c varient de 1 à 8, nous pouvons en déduire les déclarations de nos trois tableaux :

```
ligne type tableau [ [1,8] ] de booléen
diag1 type tableau [ [12,16] ] de booléen
diag2 type tableau [ [-7,7] ] de booléen
```

Pour enregistrer une reine en position (l,c), il suffit d'écrire :

```
solution[c] ← l
ligne[l] ← diag1[l+c] ← diag2[l-c] ← faux
```

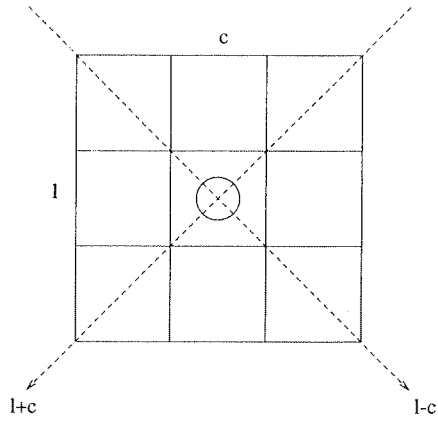


Figure 24.1 – *Les deux diagonales issues de la position (l,c).*

Et pour annuler une reine qui était en position (l,c) :

```
ligne[l] ← diag1[l+c] ← diag2[l-c] ← vrai
```

Enfin, pour vérifier si la position (l,c) est correcte devient évident :

```
ligne[l] et diag1[l+c] et diag2[l-c]
```

Nous pouvons maintenant écrire en JAVA la solution complète qui donne les quatre-vingt douze solutions de ce problème. Les déclarations des quatre tableaux sont les suivantes :

```
int [] solution = new int[8];
boolean [] ligne = new boolean[8];
boolean [] diag1 = new boolean[15];
boolean [] diag2 = new boolean[15];
```

Comme l'indice du premier élément de ces tableaux est toujours égal à zéro, le calcul d'indice pour accéder aux composants devra subir une translation égale à -1 pour les tableaux `solution` et `ligne`, égale à -2 pour le tableau `diag1`, et égale à $+7$ pour le tableau `diag2`. La méthode `essayer` s'écrit :

```
// Antécédent: c-1 reines ont correctement été placées sur les
// c-1 premières colonnes
// Rôle: essayer de placer la ce reine dans la ce colonne
void essayer(int c) {
  int i;
  for (int l=1; l<=8; l++) {
      // vérifier si on peut placer la ce reine en (l,c)
       if (ligne[l-1] && diag1[l+c-2] && diag2[l-c+7]) {
            // enregistrer la ce reine en (l,c)
             solution[c-1]=l;
             ligne[l-1] = diag1[l+c-2] = diag2[l-c+7] = false;
             if (c==8) // on a placé la huitième reine
                System.out.println(this);
```

```
            else essayer(c+1);
            // annuler la dernière reine
            ligne[l-1] = diag1[l+c-2] = diag2[l-c+7] = true;
        }
    }
} // fin essayer
```

24.3 ÉCRITURE ITÉRATIVE

Avec l'écriture itérative des algorithmes de rétro-parcours, il n'est plus possible, de fait, d'utiliser les retours d'appels récursifs pour remonter dans l'arbre de décisions. Cette version de l'algorithme s'appuie sur une procédure *régresser* dont le rôle consiste à trouver l'étape i à laquelle un nouveau candidat peut être proposé. Si une telle étape ne peut être trouvée, elle signale une impasse. Son algorithme s'exprime plus formellement :

```
Algorithme régresser(i, impasse)
    impasse ← faux
    tantque candidat à l'étape i = dernier candidat possible faire
        i ← i-1
    fintantque
    si i=0 alors impasse ← vrai
       sinon
           passer au candidat suivant de l'étape i
    finsi
```

L'algorithme itératif de rétro-parcours qui recherche une solution possible est donné ci-dessous :

```
initialisation de la solution partielle à vide
initialisation des différentes possibilités
impasse ← faux
étape ← 0
répéter
    {passer à l'étape suivante}
    i ← i+1
    étendre(i)
    vérifier(i,correcte)
    tantque non (correcte ou impasse) faire
        régresser(i,impasse)
        si non impasse alors
            vérifier(i,correcte)
        finsi
    fintantque
    {impasse ou solution à l'étape i correcte}
jusqu'à impasse ou fini(i)
si impasse alors
    pas de solution
  sinon on a trouvé une solution particulière
finsi
```

Pour obtenir toutes les solutions possibles, l'algorithme doit poursuivre son parcours de l'arbre de décisions, en forçant la régression après la découverte d'une solution. Le parcours s'achève lorsque l'impasse finale est atteinte. L'algorithme itératif qui donne toutes les solutions est le suivant :

```
initialisation de la solution partielle à vide
initialisation des différentes possibilités
impasse ← faux
i ← 0
correcte ← vrai
répéter
    {la solution partielle à l'étape i est correcte}
    si correcte alors
       si fini(i) alors
            {on a trouvé une solution}
            écrireSolution
            régresser(i,impasse)
          sinon
              i ← i+1
              étendre(i)
       finsi
    sinon {la solution partielle n'est pas correcte}
          régresser(i,impasse);
    finsi
    si non impasse alors vérifier(i,correcte) finsi
jusqu'à impasse
```

24.4 PROBLÈME DES SOUS-SUITES

Ce problème consiste à construire une suite de n éléments, pris dans un ensemble de m valeurs, telle que deux sous-suites adjacentes de soient jamais égales. Par exemple, $< 1\ 2\ 1\ 3 >$ et $< 2\ 3\ 2\ 1 >$ sont de telles suites de longueurs 4 construites sur l'ensemble $\{1\ 2\ 3\}$.

Ce problème est résolu avec un algorithme de rétro-parcours, et nous donnerons sa version itérative. Les valeurs des éléments de la suite sont des entiers positifs inférieurs à une valeur `valeurMax` et la longueur de la suite est égale à `longSuite`. Le nombre d'étapes pour atteindre une solution est donc au plus égale à `longSuite`. La solution est représentée par un tableau d'entiers et le numéro de l'étape courante sert d'indice pour accéder au dernier candidat de la solution partielle.

```
protected int[] solution;
int i;
```

La méthode `régresser` décrémente la valeur d'étape `i` tant qu'elle ne peut proposer de nouveau candidat à cette étape. Lorsque `i` est égal à zéro, l'impasse est atteinte.

```
private boolean régresser() {
  while (i!=0 && solution[i-1]==valeurMax)
         i--;
```

```
  if (i==0) return true;
  solution[i-1]++;
  return false;
}
```

La vérification de la validité de la solution partielle, c'est-à-dire vérifier si la suite ne comporte pas deux sous-suites identiques, consiste à tester toutes les sous-suites de longueur égale à un jusqu'à la moitié de la longueur et faisant intervenir le candidat choisi à l'étape $i-1$. La figure 24.2 montre la progression de la taille des sous-suites vérifiées en partant de la fin de la suite.

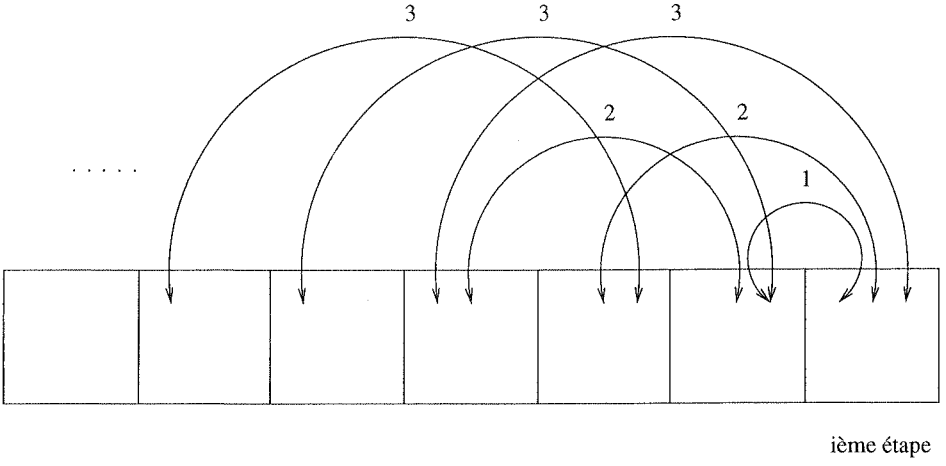


Figure 24.2 – *Vérification de la validité de la suite.*

La méthode `vérifier` est programmée en JAVA comme suit :

```
private boolean vérifier(int longueur) {
  int lgCourante=0, // longueur de la sous-suite courante
      moitié=longueur/2;
  boolean diff=true;
  while (diff && lgCourante < moitié) {
     // les sous-suites de longueurs 0 à lgCourante sont différentes
     int i=1;
     lgCourante++;
     // comparer deux sous-suites de longueur lgCourante
     do {
         diff = solution[longueur-i]!=solution[longueur-lgCourante-i];
         i++;
     } while (!diff && i!=lgCourante);
     // les sous-suites de longueur lgCourante sont différentes
     // ou bien elles sont identiques et i>lgCourante
  }
  return diff;
}
```

Enfin, la méthode `solution` qui cherche une solution particulière du problème des sous-suites est programmée comme suit :

```
public boolean solution() {
  boolean impasse=false;
  i=0;
  do {
      i++;
      // étendre la solution
      solution[i-1]=1;
      boolean correcte=vérifier(i);
      while (!(correcte || impasse)) {
           impasse=régresser();
           if (!impasse) correcte=vérifier(i);
      }
  } while (!impasse && i==longSuite);
  return !impasse;
}
```

24.5 JEUX DE STRATÉGIE

Les jeux d'échecs, de dames, ou encore le trictrac [1] sont des jeux de stratégie à deux joueurs. La programmation de ces jeux lorsque les deux joueurs sont des humains ne présentent guère d'intérêt, puisque le programme se borne essentiellement à la vérification de la validité des coups joués. En revanche, elle devient plus intéressante, lorsqu'il s'agit de faire jouer un utilisateur humain contre l'ordinateur.

Au cours d'un jeu, l'ordinateur et le joueur humain doivent, à tour de rôle, jouer un coup, le meilleur possible, c'est-à-dire celui qui conduit à la victoire finale, ou au moins à une partie nulle. La stratégie du joueur humain est basée sur son expérience du jeu ou son intuition. En général, elle tente d'imaginer une situation de jeu plusieurs coups à l'avance en tenant compte des ripostes possibles de l'adversaire. La stratégie de l'ordinateur est semblable, mais *exhaustive*. Elle consiste, à chaque étape du jeu, à essayer (récursivement) *tous* les coups possibles, alternativement de l'ordinateur et d'un joueur adverse virtuel, en ne considérant à chaque fois que les meilleurs coups de chaque camp. Cette stratégie de jeu est appelée stratégie *MinMax* [2] et nous allons voir comment l'ordinateur la met en œuvre pour jouer son meilleur prochain coup.

24.5.1 Stratégie *MinMax*

Précisons, tout d'abord, que cette stratégie ne s'applique qu'aux jeux qui s'achèvent après un nombre fini de coups, et à chaque étape, un joueur a le choix entre un nombre fini de coups possibles.

1. La version française du backgammon.

2. Proposée par O. MORGENSTERN et J. VON NEUMANN en 1945.

Pour chaque coup, la stratégie *MinMax* développe un arbre, appelé *arbre de jeu*, qui contient toutes les parties possibles à partir d'une position de jeu donnée. Chaque feuille de cet arbre correspond à un coup final d'une partie fictive, et à laquelle sont associées trois valeurs possibles : partie gagnée, nulle ou perdue. Les nœuds de l'arbre correspondent, soit à un coup joué par l'ordinateur, soit par son adversaire virtuel, et chacun d'eux contient une valeur qui représente le *meilleur* coup joué (du point de vue de l'ordinateur). La stratégie *MinMax* doit son nom au fait qu'elle cherche à maximiser la valeur des coups joués par l'ordinateur et à minimiser celle des coups joués par l'adversaire virtuel.

Lorsque l'ordinateur joue, il évalue récursivement selon la stratégie *MinMax* tous les coups possibles pour ne retenir que le meilleur. Lorsque son adversaire virtuel joue, la stratégie de l'ordinateur est de retenir le moins bon coup. Il est, par exemple, évident qu'un coup perdant pour l'adversaire, est à conserver puisqu'il conduit à la victoire de l'ordinateur. Réciproquement, un coup gagnant pour l'adversaire est à éliminer puisqu'il conduit à la défaite de l'ordinateur. Pour un nœud donné de l'arbre de jeu, la stratégie *MinMax* retourne la meilleure valeur pour l'ordinateur.

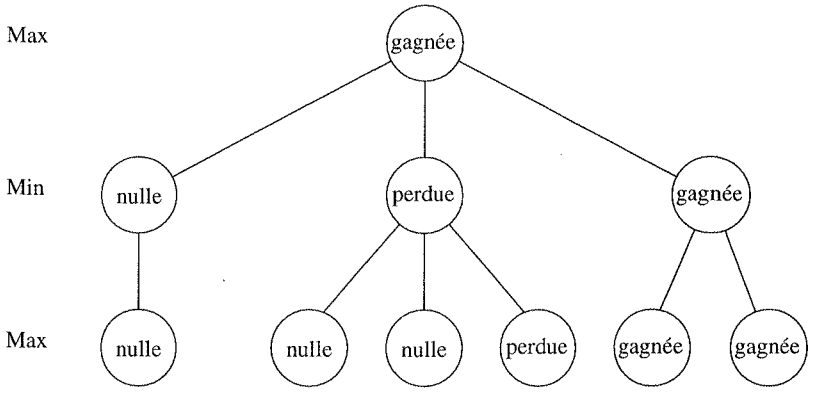


Figure 24.3 – *Un arbre de jeu.*

La stratégie *MinMax* correspond donc à un parcours en profondeur d'un arbre de jeu composé alternativement de niveaux Max et de niveaux Min (voir la figure 24.3). Les nœuds des niveaux Max sont les coups de l'ordinateur dont les valeurs sont le maximum de celles de leurs fils. Les nœuds des niveaux Min sont les coups de l'adversaire virtuel dont les valeurs sont le minimum de celles de leurs fils. La racine de l'arbre de jeu est située à un niveau Max, dont la valeur est le meilleur coup à jouer par l'ordinateur. L'algorithme suivant exprime ce parcours d'arbre.

```
Algorithme MinMax(a : arbre de jeu)
   {Rôle : parcourt en profondeur l'arbre de jeu a et retourne
           la valeur du meilleur coup à jouer par l'ordinateur}
   si feuille(a) alors
       {coup final}
       rendre valeur(a) {i.e. gagnée, nulle ou perdue}
   sinon
       si typeNoeud(a)=ordinateur alors
```

```
            {choisir la valeur maximale des fils}
            max ← Perdue
            pourtout i de 1 à longueur(forêt(a)) faire
                v ← MinMax(ièmeArbre(forêt(a),i))
                si v > max alors
                    max ← v
                finsi
            finpour
            rendre max
        sinon {l'adversaire virtuel}
            {choisir la valeur minimale des fils}
            min ← Gagnée
            pourtout i de 1 à longueur(forêt(a)) faire
                v ← MinMax(ièmeArbre(forêt(a),i))
                si v < min alors
                    min ← v
                finsi
            finpour
            rendre min
        finsi
    finsi
```

Il est important de bien comprendre que les programmes, qui mettent en œuvre cette méthode, ne construisent pas au préalable les arbres de jeu à parcourir. Ce sont les règles du jeu et la façon d'obtenir la liste des coups possibles à chaque étape qui déterminent le parcours d'un arbre de jeu *implicite*.

Nous donnons maintenant la programmation en JAVA de l'algorithme précédent. Nous considérerons qu'un coup à jouer, de type `Coup`, est formé d'une position dans le jeu et d'une valeur. La position est, par exemple, une case d'un damier ou d'un échiquier, et la valeur d'un coup est prise dans l'ensemble ordonné {Perdue, Nulle, Gagnée}. Notez qu'il est également possible de compléter, si nécessaire, ce type par la valeur d'un pion (*e.g.* un cavalier noir ou une tour blanche aux échecs). Nous définissons également un objet `jeu` qui permet d'enregistrer ou d'annuler un coup, de retourner une énumération des positions libres, d'indiquer si un coup est gagnant ou non, ou encore si la partie est dans une situation de nulle ou non. La méthode `MinMax` donnée ci-dessous tient compte de la symétrie de la méthode de jeu, ce qui permet de supprimer le test sur la nature du nœud courant. Pour cela, il suffit d'inverser la valeur du meilleur coup du joueur adverse.

```
// Antécédent : le coup final n'est pas encore trouvé
// Conséquent : le meilleur coup à jouer est retourné
Coup MinMax() {
    Coup meilleurCoup = new Coup(Perdue);
    Énumération posLibre = jeu.positionLibres();
    // essayer tous les coups disponibles possibles
    do {
        Coup coup = new Coup(Perdue, positionLibre.suivante());
        jeu.enregister(coup);
```

```
        // vérifier le coup
        if (jeu.coupGagnant(coup))
            coup.valeur=Gagnée;
        else
            if (jeu.partieNulle()) coup.valeur=Nulle;
            else  { // la partie n'est pas terminée ⇒
                    // calculer le meilleur coup de l'adversaire
                Coup coupAdversaire=MinMax();
                coup.valeur=inverserValeur(coupAdversaire.valeur);
            }
        // est-ce un meilleur coup à jouer?
        if (coup.valeur>meilleurCoup.valeur)
            // ce coup est meilleur ⇒ le conserver
            meilleurCoup=coup;
        jeu.annuler(coup);
    } while (!posLibres.finÉnumération());
    // on a trouvé le meilleur coup à jouer
    return meilleurCoup;
} // fin MinMax
```

Pour la plupart des jeux, le nombre de coups testés est très important. Pour un jeu simple comme le tic-tac-toe[3] (aussi appelé morpion), le premier coup joué par l'ordinateur nécessite de parcourir un arbre de 29 633 nœuds si le joueur humain joue son premier coup au centre de la grille, de 31 973 nœuds si son premier coup est dans un coin, et de 34 313 nœuds pour une autre case de la grille. Si l'ordinateur joue le premier, l'arbre de jeu initial possède 294 778 nœuds qu'il devra parcourir avant de jouer son premier coup[4].

Une première amélioration évidente de l'algorithme précédent est d'arrêter le parcours des fils d'un nœud lorsque la valeur maximale attendue par le meilleur coup est atteinte. Le prédicat d'achèvement de l'énoncé itératif est simplement modifié comme suit :

```
do {
    ...
} while (!posLibres.finÉnumération() && meilleurCoup.valeur!=Gagnée);
```

Avec cette modification, certains sous-arbres des arbres de jeu ne sont plus parcourus. On dit que ces sous-arbres sont coupés ou élagués. Dans le cas du tic-tac-toe, le premier coup joué par l'ordinateur ne nécessite plus que le parcours de 4 867 nœuds si le joueur humain joue son premier coup au centre de la grille, entre 2 210 et 4 872 nœuds pour les coins, et entre 7 211 et 11 172 nœuds pour les autres cases. Enfin, lorsque l'ordinateur joue le premier, 56 122 nœuds de l'arbre de jeu sont visités pour son premier coup.

3. Deux joueurs placent, alternativement, un cercle ou une croix dans les cases d'une grille 3 × 3. Le premier à aligner horizontalement, verticalement ou en diagonal, trois cercles ou trois croix a gagné.

4. En fait, l'ordinateur pourrait choisir au hasard n'importe quelle première case. Elles sont toutes équivalentes pour le premier coup.

24.5.2 Coupure α–β

La méthode de coupure α–β permet un élagage encore plus important de l'arbre de jeu, et offre une nette amélioration de l'algorithme précédent pour un résultat identique.

Considérons l'arbre de jeu donné par la figure 24.4. Les cercles contiennent des valeurs attribuées par l'algorithme aux nœuds déjà visités. Reste à parcourir le sous-arbre marqué d'un point d'interrogation. Montrons que son parcours est inutile ! Sa racine est à un niveau Min où l'algorithme minimise la valeur des nœuds (coups de l'adversaire virtuel). Sa valeur est inférieure à celles des nœuds déjà évalués au même niveau, et ne sera pas donc retenue par la racine de l'arbre de jeu qui prend la valeur maximale de ses fils. Quelle que soit la valeur obtenue par le parcours du dernier sous-arbre, celle-ci ne sera pas prise en compte. En effet, si elle supérieure à sa racine, elle est éliminée puisque sa racine conserve la valeur minimale. Si elle lui est inférieure, elle devient la nouvelle valeur de sa racine, mais demeure inférieure aux valeurs des nœuds du même niveau. Le parcours du dernier sous-arbre est donc superflu. L'élimination de ce sous-arbre dans l'algorithme *MinMax* est appelée *coupure* α. La valeur de coupure α d'un nœud n d'un niveau Min est égale à la plus grande valeur connue de tous les nœuds du niveau Max précédent. Si le nœud n possède une valeur inférieure à α alors le parcours de ses sous-arbres non parcourus est inutile.

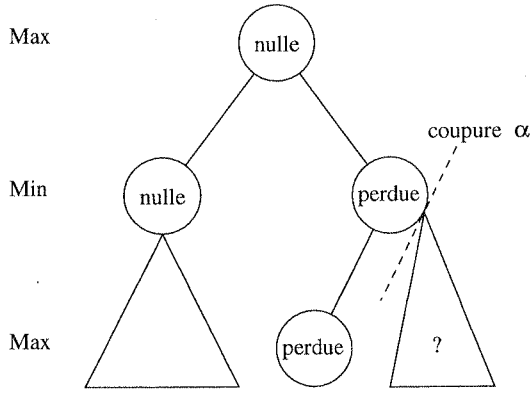


Figure 24.4 – *Coupure* α.

De façon symétrique, la figure 24.5 montre une coupure dite β. La valeur de coupure β d'un nœud n d'un niveau Max est égale à la plus petite valeur connue de tous les nœuds du niveau Min précédent. Si le nœud n possède une valeur supérieure à β alors le parcours de ses sous-arbres non parcourus est inutile.

Pour mettre en œuvre la coupure α–β, on ajoute simplement deux paramètres α et β à MinMax. Lors des appels récursifs, la valeur maximale connue du nœud ordinateur est la valeur α transmise, et la valeur minimale connue du nœud adverse est la valeur β transmise.

```
Algorithme MinMax(a : arbre de jeu, α, β)
   {Rôle : parcourt en profondeur l'arbre de jeu a et retourne
           la valeur du meilleur coup à jouer par l'ordinateur}
   si feuille(a) alors
```

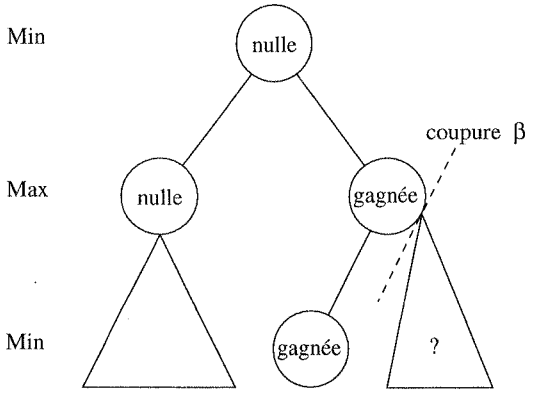


Figure 24.5 – *Coupure β.*

```
        {coup final}
        rendre valeur(a) {i.e. gagnée, nulle ou perdue}
    sinon
        si typeNoeud(a) = ordinateur alors
            {choisir la valeur maximale des fils}
            max ← α
            i ← 0
            répéter
                i ← i+1
                v ← MinMax(ièmeArbre(forêt(a),i), max, β)
                si v > max alors
                    max ← v
                finsi
            jusqu'à i = longueur(forêt(a)) ou max ⩾ β
            rendre max
        sinon {adversaire virtuel}
            {choisir la valeur minimale des fils}
            min ← β
            i ← 0
            répéter
                i ← i+1
                v ← MinMax(ièmeArbre(forêt(a),i), α, min)
                si v < min alors
                    min ← v
                finsi
            jusqu'à i=longueur(forêt(a)) ou min ⩽ α
            rendre min
        finsi
    finsi
```

Dans la méthode `MinMax` donnée ci-dessous, on conserve la symétrie en confondant les paramètres α et β en un seul dont on inverse la valeur lors de l'appel récursif.

```
// Antécédent : le coup final n'est pas encore trouvé
// Conséquent : le meilleur coup à jouer est retourné
Coup MinMax(Valeur alphabêta) {
    Coup meilleurCoup = new Coup(Perdue);
    Énumération posLibre = jeu.positionLibres();
    // essayer tous les coups disponibles possibles
    do {
        Coup coup = new Coup(Perdue, positionLibre.suivante());
        jeu.enregister(coup);
        // vérifier le coup
        if (jeu.coupGagnant(coup))
            coup.valeur=Gagnée;
        else
            if (jeu.partieNulle()) coup.valeur=Nulle;
            else  { // la partie n'est pas terminée ⇒
                    // calculer le meilleur coup de l'adversaire
                Coup coupAdversaire =
                        MinMax(inverserValeur(meilleurCoup.valeur));
                coup.valeur=inverserValeur(coupAdversaire.valeur);
            }
        // est-ce le meilleur coup à jouer?
        if (coup.valeur>meilleurCoup.valeur)
            // ce coup est meilleur ⇒ le conserver
            meilleurCoup=coup;
        jeu.annuler(coup);
    }
    while (!posLibres.finÉnumération() && meilleurCoup.valeur<alphabêta);
    // on a trouvé le meilleur coup à jouer
    return meilleurCoup;
}
```

Il a été montré que cette technique de coupure α–β limite en pratique le nombre de nœuds visités à la racine carrée du nombre de nœuds de l'arbre de jeu. Notez que les coupures de l'arbre sont d'autant plus importantes qu'un meilleur coup est trouvé rapidement, c'est-à-dire dans les sous-arbres les plus à gauche. Pour le jeu du tic-tac-toe, le premier coup joué par l'ordinateur ne nécessite plus que le parcours de 1 453 nœuds si le joueur humain joue son premier coup au centre de la grille, entre 1 324 et 2 345 nœuds pour les quatre coins, et entre 1 725 et 3 508 nœuds pour les autres cases. Enfin, lorsque l'ordinateur joue le premier, 12 697 nœuds de l'arbre de jeu sont visités pour son premier coup.

24.5.3 Profondeur de l'arbre de jeu

Les ordinateurs actuels sont capables de parcourir en une fraction de seconde l'arbre de jeu le plus grand du tic-tac-toe. Mais pour d'autres jeux, comme les échecs par exemple, le nombre de nœuds et la profondeur des arbres, c'est-à-dire celle de la récursivité, sont tels qu'un parcours jusqu'aux feuilles n'est pas praticable, même avec des coupures α–β. Pour une

partie d'échecs d'environ 40 coups, avec en moyenne 35 possibilités par coup, il y aurait 35^{80} coups à tester ! Notez également que la limitation de la profondeur des arbres de jeu peut être nécessaire avec des jeux plus simples, mais qui font intervenir plus de deux joueurs, puisque l'algorithme devra tester tous les coups des différents adversaires virtuels.

Pour ces jeux, le parcours de l'arbre est arrêté à un niveau de profondeur fixée par la puissance de l'ordinateur utilisé (taille de la mémoire centrale et rapidité du processeur). Les nœuds traités à cette profondeur sont considérés comme des feuilles et leur valeur est calculée par une fonction qui estime l'état de la partie à ce moment-là.

```
Algorithme MinMax(a : arbre de jeu, α, β, niveau)
   si niveau = 0 alors
        {profondeur maximale atteinte}
        rendre une estimation de la partie en cours
   sinon
      si feuille(a) alors
           ...
      sinon
         {a est un noeud ordinateur ou de son adversaire virtuel}
         si typeNoeud(a) = ordinateur alors
              ...
              v ← MinMax(ièmeArbre(forêt(a),i), max, β, niveau-1)
              ...
         sinon {adversaire virtuel}
              ...
              v ← MinMax(ièmeArbre(forêt(a),i), α, min, niveau-1)
              ...
         finsi
      finsi
   finsi
```

Pour conclure, ont peut dire que la stratégie *MinMax* assure nécessairement le nul ou la victoire à l'ordinateur, si le fait de débuter la partie ne donne pas un avantage irréversible à son adversaire humain. Mais, pour la plupart des jeux, comme les échecs ou les dames, la l'arbre de jeu à parcourir est trop grand et la qualité de la fonction qui estime la valeur du coup à une profondeur fixée devient essentielle.

24.6 EXERCICES

Exercice 24.1. Écrivez de façon itérative le programme des huit reines.

Exercice 24.2. Écrivez un programme qui vérifie s'il est possible de passer par toutes les cases d'un échiquier, mais une seule fois par case, à l'aide d'un cavalier selon sa règle de déplacement aux échecs.

Exercice 24.3. Modifiez le programme précédent pour qu'il recherche toutes les solutions.

Exercice 24.4. Soit un entier m et un ensemble de n entiers $E = \{x_1, x_2, \ldots, x_n\}$, cherchez un sous-ensemble de E tel que la somme de ses éléments soit égale à m.

Exercice 24.5. Soit une tôle formée d'un ensemble de carrés identiques et adjacents. On désire découper dans cette tôle un certain nombre de pièces identiques. Le découpage doit se faire sans aucune perte et en suivant uniquement les côtés des carrés qui composent la tôle.

Construire un programme qui imprime, si elle existe, une solution de découpage. Les données du programme doivent exprimer les formes et les dimensions de la tôle et des pièces. La figure 24.6 montre un exemple de tôle avec trois pièces à découper.

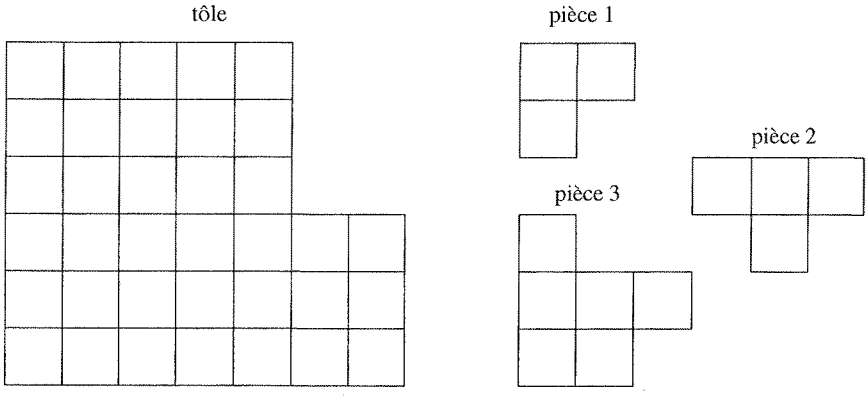


Figure 24.6 – *Une tôle et ses pièces.*

Exercice 24.6. Un labyrinthe est construit dans un carré $n \times n$. Il s'agit de trouver un chemin dans le labyrinthe qui amène d'un point de départ à un point d'arrivée. La figure 24.7 montre un labyrinthe particulier dans un carré 5×5.

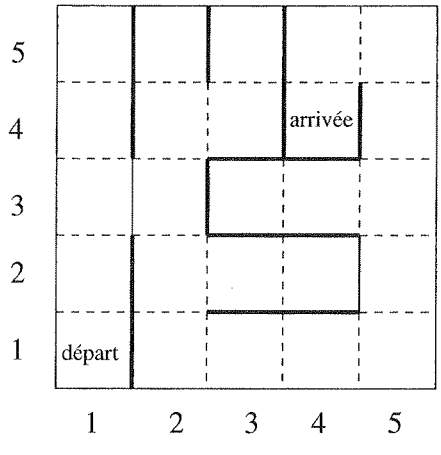


Figure 24.7 – *Un labyrinthe.*

Proposez une représentation pour le labyrinthe, puis en utilisant un algorithme de rétro-parcours, écrivez un programme qui retourne, s'il existe, le chemin entre le point de départ et le point d'arrivée.

Exercice 24.7. Programmez le jeu du tic-tac-toe. Quelle est la pronfondeur la plus grande d'un arbre de jeu ?

Exercice 24.8. Le jeu du 31 se pratique à deux joueurs et un dé. À tour de rôle, les joueurs font progresser une somme partielle par additions successives de la valeur du dé. Au début, le dé est lancé au hasard, et la somme partielle est initialisée à zéro. Ensuite, chaque joueur tourne d'un quart de tour le dé sur une des quatre faces adjacentes à celle visible du coup précédent. La nouvelle valeur du dé est ajoutée à la somme partielle. Le premier joueur qui atteint la somme 31 a gagné, celui qui dépasse 31 a perdu. Programmez ce jeu.

Exercice 24.9. Le jeu *hexxagon* se déroule sur un plateau de jeu hexagonal comprenant un nombre quadratique de cases (voir figure 24.8). Au début de la partie, chaque joueur dispose de deux pions d'une couleur donnée. Les joueurs jouent à tour de rôle. Deux types de déplacement sont possibles :

- Le clonage. Un pion peut se dupliquer sur l'une de ses six cases adjacentes à condition qu'elle soit libre.
- Le saut. Un pion peut sauter à deux cases de distance.

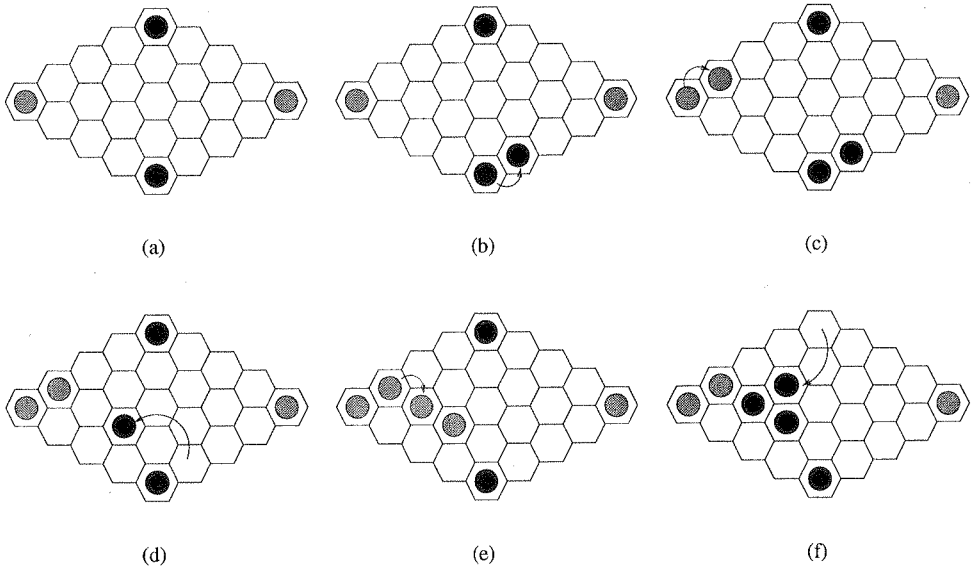


Figure 24.8 – *(a) Situation initiale; (b) clonage; (c) clonage; (d) saut; (e) clonage avec prise; (f) saut avec prise.*

Lorsqu'un pion est posé sur une case, tous les pions présents sur les six cases adjacentes prennent la couleur du pion déplacé. Le jeu se termine lorsqu'il n'y a plus de cases libres sur le plateau. Le vainqueur est le joueur qui possède le plus de pions de sa couleur.

Programmez ce jeu pour deux joueurs, l'ordinateur et un joueur humain, puis pour quatre joueurs, l'ordinateur et trois joueurs humains.

Chapitre **25**

Interfaces graphiques

Les applications interactives communiquent avec l'utilisateur au moyen d'*interfaces*. Parmi elles, les interfaces *graphiques* ont révolutionné les méthodes de dialogue avec l'ordinateur et ont simplifié son utilisation. Ce type d'interface a été élaboré dès la fin des années 60 dans les laboratoires de la compagnie XEROX, mais c'est vraiment au début des années 80 que l'interface purement graphique du système du MACINSTOSH a donné l'accès à l'ordinateur au plus grand nombre, et en particulier aux non informaticiens. Aujourd'hui, la grande majorité des systèmes d'exploitation et des applications, et particulièrement dans l'informatique individuelle, dispose d'une interface graphique.

Après avoir décrit la notion de système interactif, nous nous intéresserons plus particulièrement dans ce chapitre aux interfaces graphiques. Nous présenterons les caractéristiques principales des systèmes de fenêtrage, celles des fenêtres qu'ils manipulent et des outils de construction d'interfaces graphiques. Enfin, à travers quelques exemples simples, nous verrons les principes de base de la programmation des applications graphiques en JAVA.

25.1 SYSTÈMES INTERACTIFS

Une interface utilisateur désigne à la fois l'équipement matériel et les outils logiciels qui permettent à l'utilisateur d'assurer une communication avec l'ordinateur. Dans le passé, les ordinateurs enchaînaient l'exécution des programmes sans attente. Le mode de communication était le *traitement par lots* (mode *batch* en anglais). Aujourd'hui, la communication avec l'ordinateur est *interactive*, et l'utilisateur instaure un véritable dialogue avec l'ordinateur. Dans ces systèmes interactifs, on peut distinguer deux grands types d'interfaces : *textuelles* et *graphiques*.

Jusqu'au milieu des années 80, l'interface était essentiellement *textuelle*. Les terminaux d'accès aux systèmes exploitation étaient presque exclusivement de type alphanumérique, et ne permettaient l'affichage que d'un nombre limité de lignes de caractères pris dans le jeu ASCII. D'un point de vue logiciel, ces terminaux n'offraient donc que des interfaces textuelles, c'est-à-dire que l'utilisateur communiquait avec le système uniquement par des commandes rédigées dans un langage donné et interprétées par un interprète de commandes. Les interprètes de commandes des systèmes d'exploitation, comme par exemple un shell du système UNIX, fonctionnent selon ce principe. L'intérêt de ce type d'interface est la richesse du langage de commandes qui permet de définir ou d'inventer des comportements non prévus. En revanche, l'apprentissage du langage et la saisie des commandes est certainement une source de difficultés, particulièrement pour les non informaticiens. D'autre part, les terminaux alphanumériques n'offrent pas, en général, la possibilité de visualiser l'exécution simultanée de plusieurs tâches.

Bien que les interfaces exclusivement textuelles n'aient pas totalement disparu, aujourd'hui l'interface habituelle est *graphique*. Elle nécessite un terminal graphique, capable d'afficher une matrice de points (de l'ordre de 2048 × 2048 pour les meilleurs) et un *système de fenêtrage* à l'aide duquel on dessine des caractères, au même titre que des schémas, des images, *etc.* Un dispositif de pointage, en général une souris, est toujours associé à l'écran. Il permet de désigner ou de manipuler des parties de l'image graphique de l'écran. Pour son dialogue avec l'ordinateur, l'utilisateur se sert de la souris et accède aux fonctionnalités du système grâce à des menus déroulants ou à des boutons spécifiques. Déplacer la souris permet d'amener le pointeur sur la zone désirée de l'écran. Les combinaisons de clics simples ou clics doubles et de clics suivis d'un déplacement avec son bouton enfoncé permettent d'indiquer des actions, ou des « copier-coller ». Le système répond en affichant des menus et en faisant apparaître des fenêtres, c'est-à-dire des zones de l'écran dans des cadres qui affichent une information particulière.

Les interfaces graphiques exclusivement à base de menus déroulants, de boutons, ou de boîtes de dialogues facilitent l'interaction, mais ont des limitations. Le plus souvent, elles ne permettent qu'une utilisation passive, avec un dialogue immuable fixé par l'enchaînement des menus et des boîtes de dialogues. Elles n'offrent pas la possibilité de créer de nouveaux comportements par l'extension ou la composition de comportements existants. L'utilisation poussée à l'extrême d'une interface graphique dispense de l'emploi du clavier, mais rapidement, on s'aperçoit que l'usage exclusif de la souris devient fastidieux, et le clavier reprend ses droits. En revanche, les interfaces à *manipulation directe*, comme par exemple les éditeurs de texte ou de schémas, offrent à l'utilisateur une représentation graphique des objets informatiques et en permettent une manipulation par l'intermédiaire de la souris ou du clavier. Ce type d'interface permet une interaction *créatrice* qui n'est plus limitée par les règles fixes d'utilisation des menus déroulants ou des boîtes de dialogues.

Les interfaces des ordinateurs actuels font essentiellement appel au sens visuel de l'utilisateur (affichage sur l'écran), mais d'autres types d'interfaces permettent de combiner les autres sens et définissent de nouveaux modes de communication entre l'homme et la machine. Grâce, par exemple, à des haut-parleurs, l'utilisateur peut « entendre » l'ordinateur, ou lui « parler » à l'aide d'un micro. Il peut également sélectionner à la main des objets sur des écrans tactiles. Citons également les systèmes de réalité virtuelle qui permettent une interaction avec tous les sens de l'utilisateur dans une représentation 3D du monde. Il est aisé

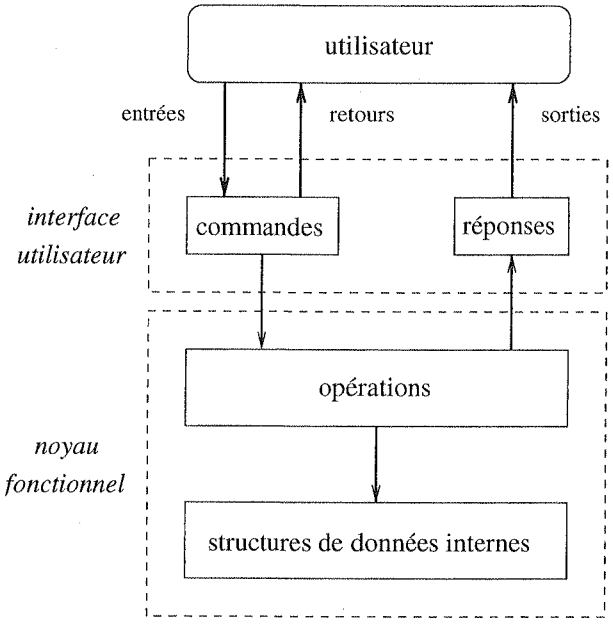


Figure 25.1 – *Structure d'une application interactive.*

d'imaginer toutes sortes de combinaisons d'interactions *multimodales*. Ces types d'interfaces sont pour l'instant beaucoup moins fréquentes que les interfaces simplement graphiques, mais vont sans aucun doute se développer avec les progrès technologiques à venir, et deviendront, peut-être, les interfaces habituelles des systèmes interactifs de demain.

25.2 CONCEPTION D'UNE APPLICATION INTERACTIVE

Une application interactive est généralement formée de deux composants principaux, l'*interface utilisateur* et le *noyau fonctionnel*. L'interface permet le dialogue entre l'utilisateur et le noyau de l'application qui assure les fonctions de calcul. L'utilisateur applique des commandes à l'aide de dispositifs d'entrée qui agissent sur le noyau fonctionnel. En retour, ce dernier produit des réponses grâce à des dispositifs de sortie. Les commandes de l'interface utilisateur exécutent des opérations internes au noyau fonctionnel qui agissent sur ses propres structures de données. L'interface offre donc à l'utilisateur une représentation des structures internes de l'application. Notez que certaines commandes de l'interface utilisateur peuvent provoquer des retours d'information sans pour autant provoquer l'exécution d'opérations du noyau. La figure 25.1 montre cette structure.

Pour illustrer ces propos, prenons l'exemple d'une interface utilisateur d'un logiciel de jeu d'échecs qui visualise graphiquement sur l'écran l'échiquier et les pièces sous forme d'icônes. Ses commandes, comme le déplacement d'une pièce sur l'échiquier à l'aide d'une souris, provoqueront l'exécution des opérations du noyau fonctionnel de contrôle de validité du coup joué, d'exécution du coup suivant selon un algorithme de coupure α-β, *etc.* Les

réponses associées aux coups joués par l'ordinateur, ou aux prises de pièces produiront en sortie des déplacements ou des suppressions de pièces à l'écran. Dans ce logiciel, un retour d'information d'une commande sera, par exemple, un changement d'apparence d'une pièce déplacée ou de son suivi à l'écran lors du déplacement, qui ne provoquera pas l'exécution d'une opération du noyau. Cette dernière ne sera, en fait, exécutée que lorsque la pièce sera effectivement posée par le joueur sur une case de l'échiquier.

Durant les deux dernières décennies, plusieurs modèles d'architecture logicielle ont été proposés pour structurer les interfaces des systèmes interactifs. Ils visent tous une séparation claire entre l'interface utilisateur et le noyau fonctionnel afin de faciliter la construction des systèmes interactifs.

Le modèle de SEEHEIM [1] [Pfa85] repose sur un modèle linguistique (lexical, syntaxique, sémantique) du dialogue homme-machine. Il définit l'interface utilisateur comme un module distinct formé de trois composants (voir la figure 25.2). Le premier appelé *présentation*, le composant lexical, s'occupe de la visualisation des objets graphiques ; le deuxième, le *contrôleur de dialogue*, le composant syntaxique, définit la structure de l'interaction entre l'utilisateur et le noyau fonctionnel ; enfin, le troisième, appelé *interface*, le composant sémantique, établit le lien avec les fonctionnalités du noyau. Ce dernier est aussi appelé adaptateur du noyau fonctionnel.

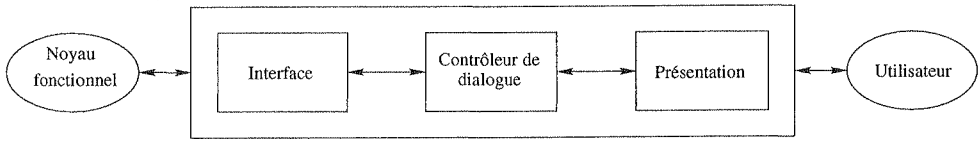


Figure 25.2 – *Le modèle de* SEEHEIM.

La caractéristique principale du modèle langage est sa forme séquentielle et centralisée des traitements. Toutefois, elle ne correspond pas forcément au comportement des utilisateurs, en particulier avec des interfaces à manipulation directe.

Un autre modèle important, le modèle multi-agents organise l'architecture de l'interface utilisateur autour d'un ensemble d'objets interactifs répartis. Plusieurs modèles ont été construits selon ce principe. Le premier d'entre eux, et le plus connu, est le modèle MVC (*Model-View-Controller*) [KP88]. Dans ce modèle, l'application interactive est construite à partir de modèles (*models*), qui sont les composants logiciels internes du noyau fonctionnel. À chaque modèle, on peut associer un ou plusieurs couples *View/Controller*. Une vue (*view*) gère la visualisation du modèle, et le contrôleur (*controller*) assure l'interface entre les modèles et les vues, auxquels il est lié, à partir des entrées de l'utilisateur. Ainsi quand ce dernier réalise une action sur un dispositif d'entrée (clavier, souris...), le contrôleur en informe le modèle associé qui modifie ses structures de données. Le modèle notifie en retour son changement d'état à ses différents contrôleurs et vues qui feront leurs propres changements d'états qui s'imposent (voir la figure 25.3). À l'origine, le modèle MVC a été mis en œuvre dans le langage à objets SMALLTALK [GR89]. On le retrouve aussi, sous des formes adaptées, dans d'autres langages, dont JAVA.

1. Il doit son nom au lieu de la conférence dans laquelle il a été proposé.

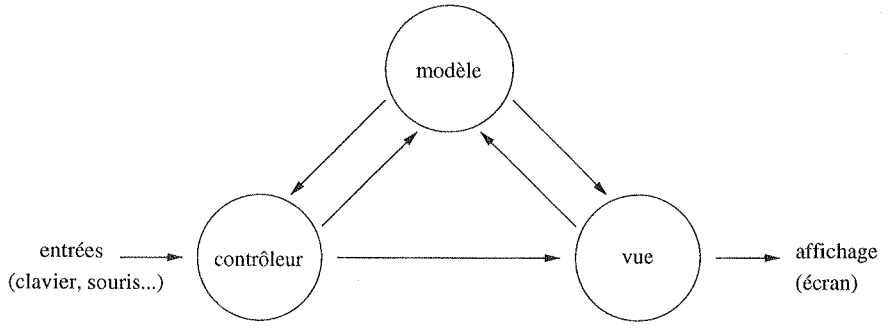


Figure 25.3 – *Le modèle MVC.*

Ces premiers modèles ont eu de nombreux successeurs comme le modèle Arch [BFL+92] qui est une extension du modèle de SEEHEIM, ou encore le modèle PAC [Cou87] qui cherche à combiner les propriétés du modèle langage et du modèle multi-agents. Le lecteur intéressé pourra se reporter à [BBL95] pour une classification des modèles existants.

L'interface utilisateur et le noyau fonctionnel sont deux parties dont les conceptions doivent être menées, autant que faire se peut, simultanément pour que les concepteurs aient une vision globale du logiciel. Mais cela n'est pas toujours possible, surtout quand il s'agit de pourvoir d'une interface un noyau fonctionnel existant.

Les spécificités des utilisateurs, surtout si ceux-ci ne sont pas des informaticiens, sont aussi des points essentiels à prendre en compte lors de la conception d'un système interactif. Les compétences, les préférences et les goûts, les possibilités humaines (ouïe, vision... mais aussi les handicaps) ou encore la psychologie des utilisateurs sont, au même titre que les contraintes techniques du noyau fonctionnel, des caractéristiques fondamentales dont le ou les concepteurs doivent tenir compte pour le développement de l'interface.

25.3 ENVIRONNEMENTS GRAPHIQUES

Les applications qui possèdent une interface graphique s'exécutent sur des ordinateurs dont le système d'exploitation est pourvu d'un *système de fenêtrage*. Dans cette section, nous décrirons tout d'abord ce qu'est un système de fenêtrage et la notion de fenêtre qu'il met en jeu. Nous présenterons ensuite les principes des outils qui servent à la construction des interfaces graphiques.

25.3.1 Système de fenêtrage

Le système de fenêtrage peut être intégré dans le système d'exploitation, comme pour les systèmes du MACINSTOSH ou de MICROSOFT, mais peut être aussi un module séparé et indépendant d'un système d'exploitation particulier, comme par exemple le système de fenêtrage X[2] basé sur la bibliothèque Xlib. Le système de fenêtrage gère l'activité d'affichage

2. Ce système de fenêtrage a été développé à l'origine, dans les années 70, au MIT.

d'une part, et les ordres d'affichage d'autre part. L'originalité du système X, puisqu'on ne la retrouve dans aucun autre système de fenêtrage, est de séparer ces deux activités selon une relation *client-serveur*. Les clients sont des applications qui assurent des calculs et qui produisent des requêtes d'affichage à destination du serveur. Le serveur est un programme qui gère le dispositif physique formé de l'écran graphique, du clavier alphanumérique et de la souris. Il traite les ordres d'affichage à l'écran et reconnaît les *événements* émis par le clavier ou la souris, et informe les clients concernés si nécessaire. Les événements qui se produisent sur un terminal sont par exemple le déplacement de la souris, l'appui sur une touche du clavier, ou un bouton de la souris, le recouvrement d'une fenêtre par une autre, *etc.* Les clients et le serveur d'affichage peuvent s'exécuter sur le même ordinateur ou sur des machines différentes communiquant alors par l'intermédiaire du réseau selon les protocoles réseaux TCP/IP ou DECnet. La propriété fondamentale du système X est d'être indépendant des ordinateurs utilisés. Seul le serveur X dépend du terminal qu'il gère. Ainsi, plusieurs clients X qui s'exécutent sur des ordinateurs différents, et même sous des systèmes d'exploitation différents, peuvent soumettre des requêtes d'affichage à un même serveur X de façon homogène. Les clients n'ont pas besoin de savoir comment fonctionne le serveur, et vice-versa. Clients et serveur respectent un protocole de communication unique et indépendant du matériel et des systèmes d'exploitation. Ce modèle client/serveur de X possède de nombreux avantages. Il permet en particulier de répartir la puissance de calcul sur plusieurs machines et de partager les ressources disponibles, alors que l'affichage a lieu sur un terminal unique. Ce terminal peut être un matériel de faible coût, alors que les clients s'exécutent sur des machines très puissantes et onéreuses.

Le système de fenêtrage gère donc des *fenêtres*, c'est-à-dire des zones de l'écran dans des cadres, la plupart du temps rectangulaires, qui affichent une information particulière des applications. Il offre tout un ensemble de primitives qui permettent aux programmeurs d'applications interactives de créer ou de détruire des fenêtres, ou encore d'en modifier le contenu. Le système de fenêtrage X ne permet pas la manipulation interactive des fenêtres par l'utilisateur, par exemple, il ne prend pas en charge les déplacements des fenêtres ou leur agrandissement. C'est le travail d'une application particulière, le *gestionnaire de fenêtres*, qui gère le dialogue avec l'utilisateur et assure généralement :

- le placement interactif des fenêtres ;
- le recouvrement et l'empilement ;
- l'icônification ;
- le *focus* du clavier ;
- le changement de taille ;
- la décoration des fenêtres ;
- les menus de commandes.

L'écran du terminal affiche des fenêtres. Même en l'absence de toute fenêtre, il en existe toujours une, la fenêtre *racine*, qui correspond à la totalité de l'écran et affiche une trame, une couleur, ou une image de fond. On l'appelle *racine* car les fenêtres créées par la suite constituent une arborescence dont le fond d'écran est la racine.

Une des premières fonctions du gestionnaire de fenêtres est la mise en place interactive des fenêtres sur la fenêtre racine. Toute nouvelle fenêtre s'affiche sur la fenêtre racine, et peut se trouver dans différents états :

- visible, icônifiée, invisible, détruite ;
- active ou inactive.

Une fenêtre correspond à un processus ; si elle est visible, les ordres d'affichage envoyés par le processus modifient ce qui s'affiche à l'écran. Une fenêtre dont on ne regarde plus le contenu occupera moins de place à l'écran si elle est *icônifiée* : dans ce cas elle est remplacée par une icône, c'est-à-dire une mini-fenêtre qui porte le même nom, mais dont on ne peut pas voir le contenu. On peut également la rendre complètement *invisible*, à condition d'avoir un moyen de la faire réapparaître ultérieurement. Enfin, une fenêtre peut être détruite. Le passage dans l'un ou l'autre de ces états se fait par des ordres au gestionnaire de fenêtres, donnés à l'aide de la souris.

Le placement des fenêtres permet le recouvrement et l'empilement des fenêtres les unes sur les autres. Les gestionnaires de fenêtres proposent en général des mécanismes, pour faire passer en avant-plan ou en arrière-plan les fenêtres.

Une seule fenêtre est *active* à la fois, c'est vers elle que tous les caractères frappés au clavier sont envoyés. On dit que cette fenêtre a le *focus*. La manière de rendre une fenêtre active est l'une des caractéristiques ergonomiques importantes de toute interface graphique. Pour certains gestionnaires de fenêtres, une fenêtre devient active simplement si le pointeur de la souris se trouve dessus ; pour d'autres, l'utilisateur doit explicitement cliquer dans la fenêtre pour lui donner le *focus*. Bien souvent, les gestionnaires de fenêtres permettent à l'utilisateur de choisir l'un ou l'autre des comportements.

L'agrandissement et le rétrécissement des fenêtres font aussi partie des tâches du gestionnaire de fenêtres. Le changement de taille se fait avec la souris ; l'utilisateur ajuste de façon interactive la taille de la fenêtre aux dimensions souhaitées. Une autre possibilité est d'agrandir à la dimension de l'écran la fenêtre, de telle façon qu'elle recouvre tout l'écran en masquant toutes les autres fenêtres.

Une fenêtre est normalement munie d'un *décor*, placé par le gestionnaire de fenêtres, qui peut servir d'une part à effectuer certaines actions sur la fenêtre. Un élément fondamental du décor est la *barre de titre*, qui comporte quelques boutons de commande, le nom de la fenêtre, et qui permet également de déplacer la fenêtre sur l'écran.

À la fenêtre racine, on associe en général des *menus*, dont les entrées permettent d'exécuter des commandes. Contrairement à l'interface classique du système du MACINSTOSH, qui place en haut d'écran une *barre de menus* dont le contenu dépend de la fenêtre active, on utilise en général avec X des *menus surgissants*, qu'on peut faire apparaître en cliquant n'importe où sur la fenêtre racine. Le minimum est d'en avoir un qui permet au moins de créer quelques nouvelles fenêtres, d'appliquer certaines actions aux fenêtres existantes, et de terminer l'exécution du gestionnaire de fenêtres.

25.3.2 Caractéristiques des fenêtres

Une fenêtre peut être caractérisée par un certain nombre de propriétés fondamentales, que l'utilisateur peut paramétrer. Nous nous limiterons à présenter trois caractéristiques de base : la géométrie, la couleur et les polices de caractères.

➢ La géométrie

La géométrie donne les dimensions de la fenêtre et sa position sur l'écran. La largeur et la hauteur de la fenêtre sont mesurées en général en *pixels*, qui correspondent aux points lumineux de l'écran, mais aussi en caractères pour certaines applications. La taille réelle de la fenêtre dépend de la distance entre deux points lumineux, et plus précisément de sa résolution, c'est-à-dire du nombre de pixels par ligne et par colonne. Plus la résolution est grande, plus la fenêtre apparaîtra petite, et plus la précision sera grande.

La position d'une fenêtre à l'écran se fait par un système de coordonnées, spécifiant des distances par rapport à un ou plusieurs repères. En général, le coin en haut à gauche d'une fenêtre possède la coordonnée (0,0).

➢ La couleur

Les écrans graphiques utilisés actuellement comprennent en chaque pixel trois types de phosphore, qui émettent respectivement les couleurs rouge, vert et bleu. L'addition de ces trois couleurs avec la même intensité donne du blanc, l'absence de ces trois couleurs donne du noir. Les couleurs possibles sont notées par trois nombres, qui donnent la valeur de l'intensité pour chacune des trois couleurs de base. Sur la plupart des écrans actuels, une intensité varie entre 0 et 255, c'est-à-dire qu'elle est représentable sur un octet, et qu'une valeur nécessite un nombre de 24 bits. Il existe donc potentiellement 2^{24} couleurs différentes, c'est-à-dire exactement 16 777216.

Le contenu de l'écran est représenté par une *mémoire d'écran*, qui doit représenter la valeur de la couleur de chaque pixel. La taille de cette mémoire d'écran aura donc une incidence directe sur le nombre de couleurs possibles. Pour un écran 1600×1280, la mémoire d'écran devra être de presque 6 Mo. Lorsque la mémoire d'écran est de taille réduite[3], le système de fenêtrage ne représente dans cette mémoire que un ou deux octets au lieu de trois, ce qui ne permet pas de coder directement la valeur de la couleur. Ce codage est fait de manière indirecte, par une *table des couleurs* associée à la mémoire d'écran, qui comprend 256 mots de 24 bits lorsqu'on fait un codage sur un seul octet. La valeur associée à un pixel dans la mémoire d'écran est donc en fait un indice dans la table des couleurs (voir la figure 25.4). On garde la possibilité des 16,5 millions de couleurs, mais on ne peut en utiliser que 256 à la fois. Un système de fenêtrage permet en général de modifier très rapidement le contenu de la table des couleurs, ce qui permet d'en avoir une par fenêtre. Dans ce cas, le changement se fait avec le changement de *focus*, au détriment des couleurs des autres fenêtres.

Tout ce mécanisme constitue le modèle RGB (*Red - Green - Blue*) des couleurs, qui est un modèle *additif* assez intuitif. Les coloristes utilisent plus souvent un modèle *soustractif*, plus proche de ce qu'on fait avec une boîte de peinture, et qui utilise comme couleurs fonda-

3. Les cartes graphiques actuelles proposent des tailles de mémoire toujours plus grandes.

mentales le jaune, le pourpre et le bleu sombre : c'est le modèle YMC (*Yellow - Magenta - Cyan*). Un autre modèle encore utilisé, le modèle HIS (*Hue - Intensity - Saturation*), dénote une couleur par sa teinte qui un est un angle sur un cercle de couleurs possibles, son intensité, qui est une valeur entre le noir et le blanc, et sa saturation, qui va de l'absence de couleur à la couleur pure.

Tous ces modèles sont équivalents, et on a par exemple, les couleurs suivantes dans le modèle RGB (notées en hexadécimale) :

noir	`000000`
blanc	`FFFFFF`
rouge	`FF0000`
vert	`00FF00`
bleu	`0000FF`
jaune (rouge + vert)	`FFFF00`
cyan (vert + bleu)	`00FFFF`
magenta (rouge + bleu)	`FF00FF`

Quand on veut choisir une couleur, on peut le faire en donnant sa valeur numérique dans le modèle RGB. Bien souvent, on dispose également d'une table de noms de couleur qui fournit des noms plus parlants : Black, Red, ou encore WhiteSmoke, MintCream, navy blue.

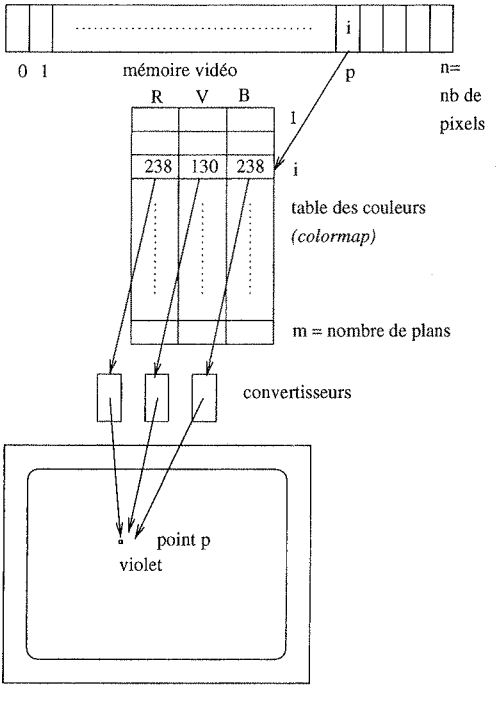


Figure 25.4 – *Affichage d'un point sur 1 octet.*

➢ Les polices de caractères

Une police de caractères est un alphabet formé de signes de même style, forme, corps, *etc.* Tout texte affiché sur l'écran nécessite l'utilisation d'une ou plusieurs polices de caractères, puisque le simple fait de mettre un mot en gras ou en italique nécessite de changer de police. Une police est caractérisée par un grand nombre d'attributs. Par exemple, dans le système X, on trouve les attributs suivants :

- le fournisseur de la police ;
- la famille ;
- la « graisse » : maigre, médium ou gras ;
- l'inclinaison : roman (droit), oblique, italique ;
- la largeur ; normal, large étroit ;
- la taille en pixel (hauteur de la partie centrale des caractères) ;
- la taille en points ;
- la résolution ;
- l'espacement : fixe ou proportionnel ;
- la largeur moyenne ;
- la norme de l'alphabet ;
- la variante de la norme.

Il est important d'avoir du discernement dans le choix des polices, pour obtenir une bonne lisibilité tout en utilisant efficacement la surface de l'écran.

25.3.3 Boîtes à outils

La programmation d'une interface graphique avec la bibliothèque d'un système de fenêtrage, comme par exemple la bibliothèque Xlib du système X, est une tâche réellement fastidieuse ; un peu comme programmer une grosse application en langage d'assemblage. Pour s'en convaincre, il suffit de lire la documentation de cette bibliothèque graphique.

Pour faciliter la programmation des interfaces graphiques, les *boîtes à outils* proposent des composants graphiques de haut niveau, appelés *widget* (*window object*) et un ensemble de fonctions pour les manipuler. Les fonctions de base sont celles de création et destruction, de placement, et de communication avec le noyau fonctionnel de l'application.

La construction de l'interface graphique se fait par assemblage de widgets organisés de façon hiérarchique. Dans toutes les boîtes à outils, on retrouve des primitives de création de widgets *simples* comme les boutons, les étiquettes, les ascenseurs et de widgets *composés* comme les menus, les boîtes de dialogues, les radio-boutons ou encore des widgets spécialement conçus pour l'assemblage de widgets simples ou composés.

Dans bon nombre de boîtes à outils, l'assemblage des widgets se fait par l'intermédiaire de widgets spécialisés selon plusieurs méthodes de placement. On en distingue en général trois qui permettent un placement des widgets de façon indépendante de leur taille effective. La première consiste à placer les widgets à l'aide d'un système de coordonnées, horizontales et verticales, sur une grille. La seconde permet de positionner les widgets les uns par rapport

aux autres. Les widgets sont empilés ou juxtaposés dans des conteneurs verticaux ou horizontaux, ou encore placés relativement les uns par rapport aux autres à l'aide de directives de type *au-dessus*, *au-dessous*, *à gauche* ou *à droite*. Enfin, la troisième consiste à spécifier un ensemble de contraintes, définies par des équations, que doivent vérifier les widgets. Ce sont, par exemple, des contraintes de distance que doivent respecter les widgets entre eux. Notez que ce dernier type de placement, contrairement aux deux premiers, n'existe que dans peu de boîtes à outils.

Les widgets offrent à leurs utilisateurs un *look and feel*, c'est-à-dire une apparence et un comportement. L'apparence concerne, par exemple, la forme ou la couleur du composant graphique. Bien souvent, l'apparence peut être paramétrée. Il sera alors possible de changer le texte inscrit sur une entrée de menu ou de modifier la taille d'un bouton. Le comportement d'un widget définit son fonctionnement qui, en général, ne peut être modifié par le programmeur. Chaque widget possède un comportement qui lui est propre en réaction aux événements auxquels il est soumis. Ainsi, pour un bouton, le fait de cliquer dessus change son apparence graphique, et par un effet d'ombre donne l'impression qu'il est enfoncé.

Contrairement aux applications des chapitres précédents, dont l'exécution suivait l'ordre séquentiel et immuable défini par leur algorithme, les programmes contrôlés par une interface graphique sont dirigés par l'utilisateur, et plus précisément par les événements auxquels réagissent les composants graphiques. On parle de *programmation dirigée par les événements*. Toutes les boîtes à outils fournissent des mécanismes qui permettent aux widgets de l'interface graphique de communiquer avec le noyau de l'application. Il existe trois grands mécanismes : les fonctions de rappels, les événements et les variables actives :

- Les fonctions de rappel (en anglais *callbacks*) sont le mécanisme le plus classique dont l'intérêt principal est la simplicité. Ces fonctions sont associées aux widgets lors de leur création (ou bien par une primitive spéciale). Leur programmation effectue des actions sur les structures de données du noyau, ou sur des widgets de l'interface graphique, par l'intermédiaire de paramètres fixés par la boîte à outils. Ainsi, lorsqu'un widget est activé, par exemple lors d'un clic sur un bouton ou sur une entrée de menu, sa fonction de rappel est exécutée. Notez que la notion d'événement n'apparaît pas explicitement dans ce mécanisme. L'inconvénient des fonctions de rappel est de destructurer le code du programme. En effet, dans la mesure où les appels de ces fonctions ne sont pas explicites, leur texte peut être placé n'importe où dans le programme, sans cadre syntaxique ou sémantique spécifique, ce qui, lorsque leur nombre devient important, entraîne un manque de lisibilité du programme global. Le lecteur désireux de mettre en œuvre ce mécanisme pourra s'exercer avec la boîte à outils *libsx* [Gia91] dont l'utilisation est particulièrement aisée dans l'environnement X.
- Les boîtes à outils qui utilisent le mécanisme d'événement, comme par exemple AWT de JAVA, définissent explicitement cette notion. La boîte à outils définit aussi la notion d'auditeurs qui sont des gestionnaires d'événements associés aux widgets. Les événements sont représentés par des objets qui sont créés lors de l'activation d'un widget et émis à destination de son auditeur associé. Les auditeurs contiennent les fonctions de rappel à exécuter lorsqu'ils interceptent un événement issu de composant graphique auquel ils sont liés. En général, l'événement est transmis en paramètre de la fonction de rappel exécutée. L'intérêt de cette méthode par rapport à la précédente est de permettre une meilleure localisation du code des fonctions de rappel.

– Les variables actives sont des variables du noyau fonctionnel qui sont associées aux widgets. Lorsqu'une variable active change de valeur, l'état du composant graphique change pour refléter la nouvelle valeur. Réciproquement, lorsque l'état du widget est modifié, une nouvelle valeur sera affectée à sa variable active exprimant le nouvel état. Ce mécanisme est très utilisé dans la boîte à outils Tk avec le langage de script Tcl [Ous94].

25.3.4 Générateurs

Même avec une boîte à outils, le développement d'une interface graphique reste une tâche ardue. Les *générateurs* aident à la conception et à l'implémentation des interfaces utilisateurs.

Les premiers générateurs ont été conçus pour produire des interfaces à partir d'une spécification exprimée sous forme textuelle dans une notation formelle (grammaires hors contexte, langages déclaratifs spécialisés, réseaux de Pétri, *etc*) la connaissance de cette notation formelle devait s'ajouter à celle du langage cible d'écriture du système interactif.

Leurs successeurs construisent la spécification de l'interface par manipulation directe, évitant ainsi au concepteur la connaissance et la programmation de la notation formelle. Ces générateurs deviennent eux aussi des outils graphiques et interactifs. Leurs éditeurs permettent à l'utilisateur de placer facilement avec la souris les composants graphiques (boutons, menus, *etc*) la future interface. S'ils facilitent la conception des interfaces par un assemblage interactif des objets graphiques, ce type de générateurs n'offre en général qu'une aide limitée quant à la programmation de leur comportement dynamique et leurs liens avec le noyau fonctionnel de l'application.

La dernière génération de ces systèmes permet le développement des applications par assemblage de composants logiciels du noyau de l'application même. Dans le monde JAVA, ces composants appelés Beans[4], éventuellement écrits et compilés séparément, sont assemblés de façon interactive et graphique à l'aide d'une plate-forme d'assemblage (*Bean Builder*). Celle-ci offre une palette de composants graphiques et des mécanismes d'assemblage qui gèrent le comportement dynamiquement de l'application.

25.4 INTERFACES GRAPHIQUES EN JAVA

La plupart des langages de programmation possède des bibliothèques de composants graphiques prêts à l'emploi pour construire des interfaces graphiques. Le langage JAVA en propose deux, AWT (*Abstract Window Toolkits*) et Swing, la seconde étant construite à partir de la première propose une version adaptée du modèle MVC. Les exemples qui suivent sont programmés avec AWT. Leur but n'est pas de faire une présentation exhaustive de cette boîte à outils graphique, mais simplement d'illustrer les notions présentées dans les sections précédentes.

4. Leur définition suit des règles spécifiques.

25.4.1 Une simple fenêtre

Pour commencer, nous donnons ci-dessous le texte d'une classe qui affiche « Bonjour à tous » dans une fenêtre graphique. Une fenêtre est une portion de l'écran, en général, rectangulaire, qui possède sur sa partie supérieure une barre de titre.

```
import java.awt.*;
public class FenêtreBonjour extends Frame {
    // les dimensions de la fenêtre
    static final int LARGEUR=300;
    static final int HAUTEUR=200;

    public FenêtreBonjour() {
       // mettre un titre à la fenêtre
        super("Bonjour");
        // fixer sa dimension
        setSize(LARGEUR,HAUTEUR);
        // la rendre visible
        setVisible(true);
    }
    public void paint(Graphics g) {
        // Affichage du message dans la fenêtre
        g.drawString("Bonjour à tous", 110, 110);
    }
} // fin classe FenêtreBonjour
```

AWT est un paquetage de l'API JAVA qui propose toutes les classes nécessaires à la construction de l'interface graphique. Dans le code précédent, la directive d'importation permet un accès direct aux classes de ce paquetage, et doit normalement apparaître en tête. La classe `Frame` définit une simple fenêtre graphique avec une bordure et une barre de titre qui peut être affichée à l'écran et dans laquelle on pourra écrire, dessiner ou placer des composants graphiques.

La fenêtre graphique que nous voulons afficher possède toutes les propriétés d'un objet `Frame` avec un message à l'intérieur. La classe `FenêtreBonjour` hérite donc de la classe `Frame`. Son constructeur exécute d'abord celui de sa super-classe pour donner un titre qui s'affichera sur la barre de la fenêtre graphique. Puis, il exécute la méthode `setSize` pour fixer les dimensions de la fenêtre. L'unité de mesure est le *pixel*, qui correspond à un point lumineux de l'écran. La taille réelle de la fenêtre dépend de la distance entre deux points lumineux, et plus précisément de sa *résolution*, c'est-à-dire du nombre de pixels par ligne et par colonne. Plus la résolution est grande, plus la fenêtre apparaîtra petite, et plus la précision sera grande. Dans notre exemple, la fenêtre est donc un rectangle de largeur 300 pixels et de hauteur 200 pixels. Enfin, le constructeur exécute la méthode `setVisible` avec comme paramètre **`true`** pour afficher la fenêtre à l'écran. Notez qu'une fenêtre graphique peut exister, sans pour autant être visible sur l'écran. Voilà pourquoi l'appel à `setVisible` est nécessaire. La méthode `paint` n'est appelée nulle part dans le programme. C'est l'environnement AWT qui s'en charge automatiquement à la création de la fenêtre ou lors de sa désicônification. Son paramètre de type `Graphics` est la zone graphique de la fenêtre dans laquelle la méthode `drawString` écrira le message. Les coordonnées du point de départ du message sont (110,110). Le point (0,0) est l'angle supérieur gauche. La création d'un objet de type `FenêtreBonjour` provoque la création d'une fenêtre semblable à celle de la figure 25.5.

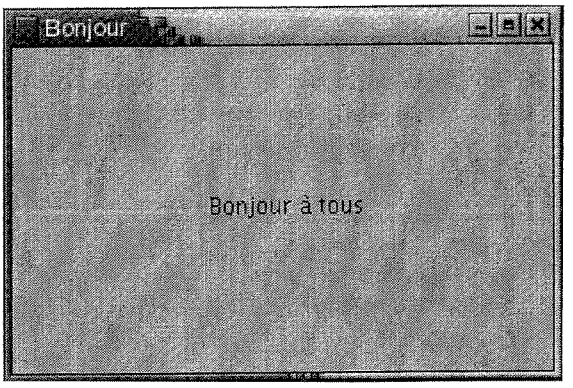


Figure 25.5 – *Une simple fenêtre.*

Dans AWT, un événement est un objet. Tous les composants graphiques peuvent émettre des événements, et peuvent être aussi à l'écoute des événements grâce à des *auditeurs* (*listeners* en anglais). L'auditeur d'un objet spécifie les événements qu'il désire écouter, et contient des fonctions de rappel à exécuter, appelées *gestionnaires d'événements*.

Nous allons maintenant faire réagir cette fenêtre à un événement extérieur. Lorsqu'elle possède le focus, nous voulons que cette fenêtre se ferme et que l'application s'achève lorsque l'utilisateur appuie sur la touche « q » de son clavier. La classe `FermetureFenêtre` qui suit fabrique par héritage de la classe abstraite `KeyAdapter` l'auditeur chargé de cette tâche. La classe `KeyAdapter` traite tous les événements issus des touches du clavier. Parmi les méthodes abstraites de cette classe, nous allons définir `keyPressed` dans l'auditeur. Son paramètre de type `keyEvent` permet de tester la touche qui a été appuyée grâce à la méthode `getKeyChar()`. Les classes qui gèrent les événements appartiennent au paquetage `java.awt.event`.

```
import java.awt.event.*;
public class FermetureFenêtre extends KeyAdapter {
    public void keyPressed(KeyEvent e) {
           if (e.getKeyChar() == 'q')
                System.exit(0);
    }
} // fin classe FermetureFenêtre
```

La méthode `addKeyListener` permet d'ajouter à une fenêtre des auditeurs qui traitent la réception des événements issus des touches du clavier. Pour ajouter l'auditeur précédent, il suffit de placer dans le constructeur de la classe `FenêtreBonjour` l'instruction suivante :

```
addKeyListener(new FermetureFenêtre());
```

Afin de contracter le code, notez que l'auditeur peut être également défini par une classe *anonyme* déclarée au moment de sa construction.

```
addKeyListener(new KeyAdapter() {
        public void keyPressed(KeyEvent e) {
            if (e.getKeyChar() == 'q') System.exit(0);
        }});
```

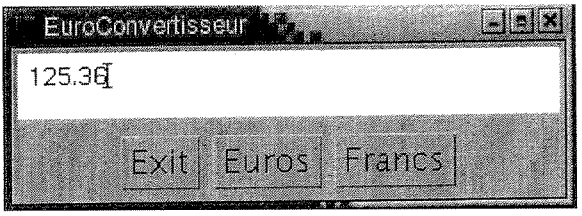


Figure 25.6 – *Un convertisseur d'euros.*

25.4.2 Convertisseur d'euros

On se propose de programmer un convertisseur d'euros en francs (et inversement). Le cahier des charges de ce convertisseur est simple : un utilisateur pourra saisir une somme d'argent que le convertisseur convertira soit en francs soit en euros suivant le taux de conversion en vigueur. Il pourra aussi mettre fin à l'application à tout moment.

Avant de programmer l'interface graphique de ce convertisseur, donnée par la figure 25.6, nous allons d'abord nous intéresser à son noyau fonctionnel. Les fonctions de conversion sont au cœur de celui-ci. Ce noyau fonctionnel est représenté par la classe `Conversion` suivante :

```
public class Conversion {
    // Le taux de conversion officiel : 1 euro = 6.55957 francs
    static final double TAUX_DE_CONVERSION = 6.55957;

    // Rôle : convertir des francs en euros
    public static double convertirEnEuros(double francs) {
        return francs / TAUX_DE_CONVERSION;
    }
    // Rôle : convertir des euros en francs
    public static double convertirEnFrancs(double euros) {
        return euros * TAUX_DE_CONVERSION;
    }
} // fin classe Conversion
```

Pour créer l'interface graphique, nous allons assembler des composants graphiques prédéfinis par la boîte à outils. Nous avons besoin d'une entrée pour la saisie de la somme d'argent, et de trois boutons pour les conversions et l'achèvement de l'application. Pour disposer ces composants graphiques à l'intérieur d'une fenêtre graphique comme sur la figure 25.6, AWT propose différents systèmes d'agencement définis par les classes `FlowLayout`, `BorderLayout`, `GridLayout` ou encore `GridBagLayout`. La méthode `setLayout` de la classe `Container` (dont hérite la classe `Frame`) permet de fixer le type d'agencement désiré. Ensuite, les appels successifs à la méthode `add` permettent de placer les composants selon le mode d'agencement choisi.

La programmation en JAVA de l'interface graphique du convertisseur est donnée plus loin. Les quatre widgets de l'interface sont déclarés comme attributs de la classe `EuroConvertisseur` qui hérite de `Frame`. L'entrée possède une largeur de 25 caractères au maximum, sans texte initial. Chacun des trois boutons est créé avec une étiquette qui

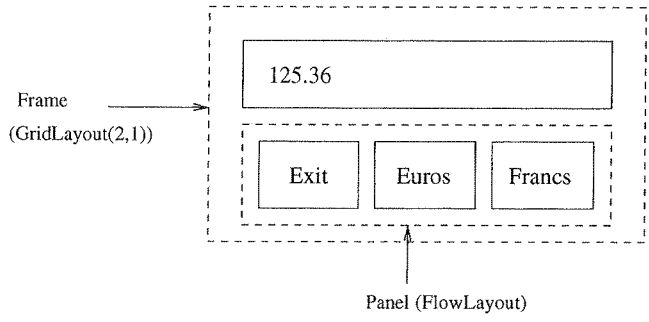


Figure 25.7 – *Agencement des composants graphiques.*

l'identifie. Pour l'agencement de ces widgets, nous allons utiliser la classe `GridLayout` qui dispose les widgets sur une grille de dimensions données, et la classe `FlowLayout` qui les dispose de façon régulière de gauche à droite et du haut vers le bas. La fenêtre principale est une grille à deux lignes et une colonne. L'entrée de type `TextField` est placée sur la première ligne. Les trois boutons sont placés dans la seconde ligne. Pour cela, ils sont regroupés dans un objet de type `Panel`. Cette classe facilite la construction hiérarchique des interfaces graphiques, puisque chaque *panel* pourra être décrit séparément et pourra posséder son propre système de coordonnées et son propre système d'agencement. La figure 25.7 montre l'organisation de l'interface graphique avec ces deux systèmes d'agencement. Enfin, la fenêtre est rendue visible, après que sa taille définitive en fonction de celle de ses composants a été calculée grâce à la méthode `pack`.

```
import java.awt.*;
public class EuroConvertisseur extends Frame {
    TextField monnaie = new TextField("", 25);
    Button euros  = new Button("Euros"),
           francs = new Button("Francs"),
           exit   = new Button("Exit");
    public EuroConvertisseur() {
        // donner un titre à la fenêtre
        super("EuroConvertisseur");
        // Agencement principal : grille 2x1
        setLayout(new GridLayout(2,1));
        // positionner l'entrée avec un fond blanc
        monnaie.setBackground(Color.white);
        add(monnaie);
        // placer côte à côte dans un panel les deux boutons
        // de conversion et celui de sortie
        Panel p = new Panel();
        // définir son système d'agencement
        p.setLayout(new FlowLayout());
        p.add(exit);
        p.add(euros);
        p.add(francs);
        // positionner le panel au-dessous de l'entrée
        add(p);
```

```
        // ajuster la taille des composants et
        // rendre visible la fenêtre
        pack();
        setVisible(true);
    }
} // fin classe EuroConvertisseur
```

L'interface précédente ne réagit qu'à la seule insertion de caractères dans l'entrée. C'est le comportement par défaut du widget `TextField`. Si l'utilisateur appuie sur l'un des boutons, son apparence change, et l'événement `ActionEvent` est généré. La méthode `actionPerformed` est alors automatiquement exécutée dans l'auditeur associé au bouton. Cet auditeur doit implémenter l'interface `ActionListener` et définir la méthode `actionPerformed`. Dans notre application, nous placerons cette méthode dans la classe `EuroConvertisseur`. À l'aide, par exemple, de la méthode `getSource` de la classe `ActionEvent`, elle devra déterminer sur quel bouton l'utilisateur a cliqué. Cette méthode retourne une référence sur le bouton qui a été enfoncé et il suffit donc de tester sa valeur pour exécuter l'achèvement de l'application, ou pour remplacer la valeur réelle contenue dans l'entrée par sa conversion en francs ou en euros.

```
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == exit) System.exit(0);
    // sinon traiter la conversion
    double valeur=0;
    // prendre le contenu de l'entrée
    try { valeur = (Double.parseDouble(monnaie.getText())); }
    catch (NumberFormatException e) {
        monnaie.setText("valeur␣réelle␣erronée");
        return;
    }
    // valeur est bien un réel ⇒ écrire sa conversion dans l'entrée
    if (e.getSource() == euros) // bouton euros
        monnaie.setText(
                Double.toString(Conversion.convertirEnEuros(valeur)));
    else // bouton francs
         monnaie.setText(
              Double.toString(Conversion.convertirEnFrancs(valeur)));
}
```

L'association d'un auditeur à un bouton se fait avec la méthode `addActionListener`. Dans notre exemple, et puisqu'elle possède la méthode `actionPerformed`, c'est la classe `EuroConvertisseur` qui est elle-même l'auditeur des trois boutons[5]. On applique alors la méthode `addActionListener` avec **`this`** en paramètre à chacun des boutons.

```
exit.addActionListener(this);
euros.addActionListener(this);
francs.addActionListener(this);
```

5. Notez que l'on aurait pu associer un auditeur différent à chacun des boutons. Lorsqu'il y a de nombreux boutons, cela permet d'éviter la cascade de tests pour déterminer le bouton sur lequel l'utilisateur a appuyé.

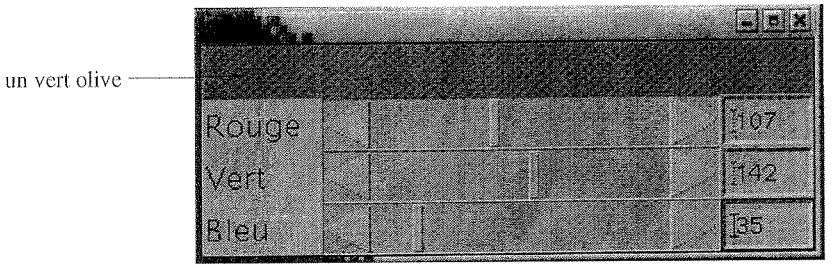


Figure 25.8 – *Composant de visualisation de couleurs.*

25.4.3 Un composant graphique pour visualiser des couleurs

Dans ce dernier exemple, nous allons concevoir un composant graphique qui pourra être utilisé par ailleurs comme n'importe quel autre composant graphique d'AWT.

Avec ce composant, que nous appellerons `CouleurGraphique`, il s'agit de créer, de modifier et de visualiser des couleurs selon le modèle RGB (*Red, Green, Blue*). Rappelons que dans ce modèle, une couleur est définie à partir de la composition de trois valeurs, comprises entre 0 et 255, représentant une valeur d'intensité des couleurs rouge, vert et bleu. Une couleur pourra être construite et modifiée à partir des constructeurs et des méthodes de la classe, ou bien de façon interactive à l'aide de l'interface graphique donnée par la figure 25.8. Les barres de défilement font varier l'intensité des trois couleurs dont les valeurs apparaissent dans les trois entrées associées. Réciproquement, la modification des valeurs d'intensité dans les entrées ajustera la position courante des barres de défilement. Enfin, quelle que soit la façon de modifier les intensités, la nouvelle couleur spécifiée est visualisée sur le haut de la fenêtre.

Le widget `CouleurGraphique` est défini comme un `Panel` afin qu'il puisse être composé avec d'autres widgets. Il est construit à partir de dix autres composants graphiques : quatre étiquettes de type `Label`, trois barres de défilement de type `Scrollbar` et trois entrées de type `TextField`. Ces widgets sont placés selon le système d'agencement `BorderLayout` qui organise le panel. Les trois étiquettes Rouge, Vert et Bleu sont placées à gauche (`BorderLayout.EAST`), les trois barres de défilement horizontales au centre (`BorderLayout.CENTER`), les trois entrées à droite (`BorderLayout.WEST`), et enfin l'étiquette qui visualise la couleur est positionnée au-dessus (`BorderLayout.NORTH`).

Les barres de défilement sont orientées de façon horizontale, graduées de 0 à 256, initialisée à 0 et la taille de leur curseur est d'une unité. En fait, la valeur maximale atteinte sera 255 (256 - la taille du curseur). Les trois entrées possèdent également une valeur initiale égale à zéro.

```
public class CouleurGraphique extends Panel {
    // les trois entrées pour les intensités rouge, vert et bleu
    // initialisées à la valeur 0
    TextField tfR = new TextField("0",3),
              tfV = new TextField("0",3),
              tfB = new TextField("0",3);
    // les trois barres de défilement pour les intensités
    Scrollbar sbR = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 256),
```

```
            sbV = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 256),
            sbB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 256);
    Label labRVB = new Label(), // pour visualiser la couleur choisie
          labR = new Label("Rouge"),
          labV = new Label("Vert"),
          labB = new Label("Bleu");

    public CouleurGraphique () {
        setLayout(new BorderLayout());
        // les étiquettes Rouge, Vert, Bleu à gauche
        Panel p1 = new Panel();
        p1.setLayout(new GridLayout(3,1));
        p1.add(labR);
        p1.add(labV);
        p1.add(labB);
        add(p1,BorderLayout.WEST);
        // les barres de défilement au centre
        Panel p2 = new Panel();
        p2.setLayout(new GridLayout(3,1));
        p2.add(sbR);
        p2.add(sbV);
        p2.add(sbB);
        add(p2, BorderLayout.CENTER);
        // les entrées à droite
        Panel p3 = new Panel();
        p3.setLayout(new GridLayout(3,1));
        p3.add(tfR);
        p3.add(tfV);
        p3.add(tfB);
        add(p3,BorderLayout.EAST);
        // l'étiquette qui visualise la couleur au-dessus
        add(labRVB, BorderLayout.NORTH);
    }
} // fin classe CouleurGraphique
```

Il s'agit maintenant de relier ces composants graphiques à une couleur. Pour cela, nous déclarons trois attributs entiers qui représenteront les trois intensités de la couleur courante. Ces attributs sont en fait des variables actives telles que leur modification entraîne la mise à jour des widgets auxquels ils sont reliés ; réciproquement toute modification des widgets entraîne un changement de valeur de ces attributs. Le code suivant est inséré dans la classe `CouleurGraphique`. On déclare trois entiers `rouge vert` et `bleu` et on modifie le constructeur de telle façon à les initialiser à l'aide des méthodes de changement de valeur d'intensité (décrites plus loin), qui provoquent également la visualisation de la couleur dans l'étiquette du haut.

```
int rouge, vert, bleu;
public CouleurGraphique (int r, int v, int b) {
    ...
    add(labRVB, BorderLayout.NORTH);
    changerRouge(r); changerVert(v); changerBleu(b);
}
public CouleurGraphique () { this(0, 0, 0); }
```

Lorsqu'on modifie une des intensités de la couleur courante, ce changement doit se répercuter sur la barre de défilement et l'entrée qui lui correspondent, ainsi que sur l'étiquette de visualisation du haut. C'est par exemple le rôle de la méthode `changerRouge` donnée au-dessous. Tout d'abord, elle mémorise la nouvelle intensité rouge, puis l'affecte à la barre de défilement (ce qui a pour effet de déplacer la position de son curseur) et à l'entrée associée. Une nouvelle couleur courante est alors créée en conservant les intensités verte et bleue précédentes, et on la visualise dans l'étiquette.

```
public void changerRouge(int r) {
// Rôle : change la valeur de l'intensité rouge et
// répercute la modification sur les widgets liés
     rouge = r;
     sbR.setValue(r); // ajuster la barre de défilement
     tfR.setText(String.valueOf(r)); // puis l'entrée
     // visualiser la nouvelle couleur
     labRVB.setBackground(new Color(rouge, vert, bleu));
}
```

Les deux autres méthodes de modification des intensités verte et bleue s'écrivent sur le même modèle.

Le déplacement du curseur des barres de défilement provoque l'émission d'événements `AdjustmentEvent`. La méthode `adjustmentValueChanged` est alors automatiquement exécutée dans l'auditeur associé à la barre de défilement. Dans notre exemple, l'auditeur est la classe `CouleurGraphique`. Elle implémente l'interface `AdjustmentListener` et définit la méthode `adjustmentValueChanged`. Cette dernière répercute la modification de la barre de défilement à l'aide des méthodes `changerRouge`, `changerVert` et `changerBleu` précédentes.

```
public class CouleurGraphique extends Panel
implements AdjustmentListener, ActionListener {
...
  public void adjustmentValueChanged(AdjustmentEvent e) {
      if (e.getSource() == sbR)
          changerRouge(sbR.getValue());
      else
          if (e.getSource() == sbV)
              changerVert(sbV.getValue());
          else // la barre de défilement de l'intensité bleue
              changerBleu(sbB.getValue());
  }
  ...
} // fin classe CouleurGraphique
```

L'appui sur la touche « entrée » du clavier dans l'une des entrées associées aux intensités provoque l'émission d'événements `ActionEvent`. La méthode `actionPerformed` de l'auditeur associé est exécutée. L'auditeur doit alors implémenter l'interface `ActionListener`. La classe `CouleurGraphique` est aussi l'auditeur de ces événements. Elle implémente donc `ActionListener` et définit `actionPerformed` comme suit :

```
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == tfR)
        changerRouge(Integer.parseInt(tfR.getText()));
    else
        if (e.getSource() == tfV)
            changerVert(Integer.parseInt(tfV.getText()));
        else // l'entrée de l'intensité bleue
            changerBleu(Integer.parseInt(tfB.getText()));
}
```

Pour terminer, il faut spécifier les auditeurs des événements émis par les barres de défilement et les entrées. Puisque l'auditeur qui contient les méthodes `adjustmentValueChanged` et `actionPerformed` est la classe `CouleurGraphique`, son association avec les widgets concernés se fait simplement dans le constructeur :

```
// associer l'auditeur aux barres de défilement
sbR.addAdjustmentListener(this);
sbV.addAdjustmentListener(this);
sbB.addAdjustmentListener(this);
// associer l'auditeur aux entrées
tfR.addActionListener(this);
tfV.addActionListener(this);
tfB.addActionListener(this);
```

25.4.4 Applets

Les applets sont des petites applications graphiques JAVA intimement liées au *World Wild Web*[6]. Contrairement aux applications graphiques précédentes qui s'exécutent de façon autonome, les applets nécessitent une autre application graphique pour leur exécution, un visualisateur spécialisé (*appletviewer*) ou un navigateur WWW. Leur contexte d'exécution est celui d'un document HTML interprété par le visualisateur chargé de l'exécution de l'applet. Le chargement de l'applet dans le document HTML se fait avec la commande `applet`. Chaque applet a les moyens d'accéder à des informations sur son contexte d'exécution, en particulier sur ses paramètres de chargement ou sur les autres applets du document.

Les composants graphiques habituels (boutons, labels, canevas, *etc.*) d'AWT ou Swing peuvent être utilisés pour la construction des applets, et réagissent normalement aux événements (clics de souris, etc). Les applets peuvent également traiter des images (généralement au format *gif*) et des documents sonores (généralement au format *au*), si l'environnement d'exécution dispose d'un dispositif de reproduction sonore.

6. Également appelé WWW, W3, WEB ou encore la Toile. L'idée fondamentale de WWW est de fournir à la consultation, par l'intermédiaire de serveurs spécialisés, des documents de type *hypertexte*, c'est-à-dire contenant des références, appelées *liens*, à d'autres documents, eux-mêmes consultés de la même manière. Il suffit de cliquer sur un lien pour voir apparaître la page vers laquelle pointait ce lien. Ces documents sont des fichiers rédigés dans le langage de commande HTML (*HyperText Markup Language*). Ce langage est formé d'un ensemble de commandes dispersées dans un texte ordinaire, qui permettent d'une part de placer des indications de présentation, d'autre part de placer des documents graphiques ou sonores, ou encore des programmes à exécuter, enfin de placer des références à d'autres documents. L'accès au WWW se fait à l'aide d'un navigateur (ou butineur), qui en général fournit une interface graphique.

Généralement placées sur des serveurs WWW, les applets s'exécutent localement dans le navigateur de l'utilisateur après leur téléchargement à travers le réseau. Ce type de fonctionnement impose bien sûr des règles de sécurité pour garantir l'intégrité de la machine de l'utilisateur. Par exemple, une applet ne peut, par défaut[7], lire ou écrire des fichiers ou exécuter des programmes de la machine client. L'action d'une applet est limitée par un composant spécifique du navigateur (*SecurityManager*) qui en assure le contrôle.

AppletEuroConvertisseur

À titre d'exemple, nous allons reprendre le convertisseur d'euros de la section 25.4.2 pour le transformer en applet. L'écriture d'une applet avec AWT se fait nécessairement par héritage de la classe *Applet*. L'en-tête de notre nouvelle classe *AppletEuroConvertisseur*, qui n'hérite donc plus de la classe `Frame`, aura la forme suivante :

```
import java.applet.*;
public class AppletEuroConvertisseur
extends Applet implements ActionListener {
```

La classe *Applet* fournit une interface entre l'applet et son contexte d'exécution par l'intermédiaire d'un certain nombre de méthodes. Plusieurs d'entre elles peuvent être redéfinies dans la classe héritière. En particulier les méthodes suivantes :

- `init`, appelée par le visualisateur lors du premier chargement de l'applet ;
- `start`, appelée par le visualisateur juste après `init`, ou automatiquement à chaque réapparition à l'écran (changement de page ou désicônification), pour signifier à cette applet qu'elle doit démarrer ou redémarrer sa véritable exécution ;
- `stop`, appelée par le visualisateur pour arrêter l'exécution de l'applet à chacune de ses disparitions de l'écran (changement de page ou icônification) ou juste avant sa destruction complète ;
- `destroy`, appelée par le visualisateur pour détruire toutes les ressources allouées par l'applet.

Pour notre applet, nous redéfinirons la méthode d'initialisation `init`. Celle-ci reprend les instructions du constructeur de la classe `EuroConvertisseur` qui agencent l'entrée et les boutons, et associent les auditeurs. Comme la classe `Frame`, `Applet` hérite de la classe `Container` et de sa méthode `add` pour ajouter les composants graphiques.

```
public void init() {
    setLayout(new GridLayout(2,1));
    // l'entrée
    monnaie.setBackground(Color.white);
    add(monnaie);
    // les deux boutons de conversion
    Panel p = new Panel();
    p.setLayout(new GridLayout(1,2));
    p.add(euros); euros.addActionListener(this);
    p.add(francs); francs.addActionListener(this);
    add(p);
}
```

7. Elle a la possibilité de le faire si elle possède une autorisation.

Notez la disparition du bouton `exit`, qui n'a plus de raison d'être puisqu'une applet n'est pas une application indépendante. La destruction de l'applet est gérée par son visualisateur et la méthode `destroy`. De même, les appels aux méthodes `pack` et `setVisible` deviennent inutiles puisque la gestion de l'affichage est sous le contrôle du visualisateur.

Enfin, le lancement de l'exécution de l'applet est spécifié par la commande `applet` placée dans un document HTML qui sera interprété par le visualisateur. Cette commande indique le nom du fichier compilé et les dimensions d'affichage de la fenêtre.

```
<applet code="AppletEuroConvertisseur.class"  width=400 height=150>
</applet>
```

25.5 EXERCICES

Pour les exercices qui suivent, vous aurez besoin des pages de documentation des paquetages d'AWT.

Exercice 25.1. Modifiez la classe `EuroConvertisseur` afin de visualiser simultanément la valeur à convertir et la valeur convertie.

Exercice 25.2. Renommez `DessinsGéométriques` la classe `FenêtreBonjour` de la page 365, puis modifiez sa méthode `paint` afin d'afficher dans la fenêtre des rectangles et des cercles. Utilisez les méthodes de la classe `Graphics`. Pour dessiner dans la zone graphique, on choisit d'abord une couleur grâce à la méthode `setColor`. Ensuite, on fait appel à l'une des méthodes `drawXXX` qui dessinera la figure désirée avec la couleur fixée.

Exercice 25.3. Une barre de menus est un composant qui se place au-dessus d'une fenêtre et qui contient plusieurs menus. Chaque menu apparaît à l'écran lorsque l'utilisateur clique dessus et propose plusieurs entrées. À chaque entrée est associée une commande (ou éventuellement un sous-menu). Dans AWT, un composant `MenuBar` est une barre de menus qui contient des objets `Menu` contenant eux-mêmes des objets `MenuItem`. Une barre de menus est ajoutée avec la méthode `setMenuBar`. La méthode `add` de `MenuBar` permet l'ajout de menus de type `Menu`. Les entrées d'un menu, de type `MenuItem`, sont ajoutés à l'aide de la méthode `add` de `Menu`. Enfin, la méthode `addActionListener` de `MenuItem` permet d'associer un auditeur de type `ActionListener` à une entrée.

Dans la classe `DessinsGéométriques` précédente, placez au-dessus de la fenêtre une barre de menus qui contient les menus *Gestion* et *Figures*. Dans le premier menu, ajoutez l'entrée *Quitter*, et dans le second les entrées *Rectangle*, *Carré*, *Ellipse* et *Cercle*. Puis, associez à l'entrée *Quitter* l'action d'achèvement de l'application (*i.e.* `exit`).

Exercice 25.4. La classe `Canvas` d'AWT définit une zone graphique dans laquelle il est possible de manipuler les dessins exécutés. Cette classe possède la méthode `paint`, héritée de `Component`, qui est appelée implicitement chaque fois qu'il est nécessaire de redessiner la zone graphique du canevas. Une classe héritière de la classe `Canvas` peut dès lors redéfinir la méthode `paint` et afficher toutes les informations voulues dans la zone graphique (le paramètre de type `Graphics` de `paint`.).

Écrivez une classe `Toile` héritière de `Canvas`. Ajoutez dans la fenêtre de la classe `DessinsGéométriques` de l'exercice précédent un objet de type `Toile`, puis associez

à chaque entrée du menu *Figures*, la création de la figure correspondante dans le canevas. Après chaque création, il faudra réafficher *explicitement* la zone graphique du canevas pour visualiser à l'écran la nouvelle figure. Pour cela, vous appellerez la méthode `repaint` (et non pas `paint`[8]).

Vous utiliserez les classes définies au chapitre 12 pour représenter les figures. Ajoutez dans la classe `Figure` un attribut qui définit les coordonnées d'origine de la figure. Vous devrez également mémoriser toutes les figures créées dans une table. La méthode `paint` de `Toile` parcourra cette table pour réafficher correctement toutes les figures. Pensez à munir les figures d'une méthode `dessiner`.

Exercice 25.5. L'auditeur de type `MouseListener` intercepte les événements de type bouton appuyé ou relâché, tandis que l'auditeur de type `MouseMotionListener` traite les événements liés aux mouvements du pointeur de la souris.

Complétez la classe `Toile` avec les méthodes `mousePressed` et `mouseDragged` pour sélectionner et déplacer une figure de la toile. Ces deux méthodes possèdent comme paramètre un événement de type `MouseEvent` à partir duquel on récupère les coordonnées du pointeur de souris avec les méthodes `getX` et `getY`.

On considérera que la figure sélectionnée sera celle dont l'origine est la plus proche du pointeur de la souris. Pensez à munir la classe `Figure` d'une méthode `distance` qui retourne la distance entre un point de la toile et le point d'origine de la figure courante, ainsi qu'une méthode `déplacer` qui change le point d'origine de la figure courante.

Exercice 25.6. Modifiez votre auditeur d'événements souris afin de faire varier la taille des figures lorsqu'on *tire* sur une figure avec le deuxième bouton de la souris enfoncé. La méthode `getButton` de `MouseEvent` indique le bouton enfoncé. Pensez à définir dans vos classes de figures une méthode de dilatation selon un coefficient passé en paramètre.

Exercice 25.7. À l'aide d'un objet `CouleurGraphique` (voir à la page 370), complétez la classe `DessinsGéométriques` afin de permettre le choix de la couleur des figures dessinées. Si cela est nécessaire, la classe `CouleurGraphique` pourra être étendue.

Vous possédez maintenant l'ébauche d'un éditeur graphique et interactif de formes géométriques que vous pouvez compléter à votre guise.

8. La méthode `paint` n'est jamais appelée directement. En fait, `repaint` appelle la méthode `update` qui appelle `paint`.

Bibliographie

[ACM93] ACM SIGPLAN Notices. *Conference on History of Programming Languages (HOPL-II)*, April 1993.

[AFN82] AFNOR. *Le Langage Pascal, spécification et mise en œuvre.* AFNOR, 2e édition, 1982.

[ANS76] ANSI. *American National Standard Programming Language – PL/I, ANSI X3.53*, 1976.

[ANS78] ANSI. *American National Standard Programming Language – FORTRAN.* ANSI, 1978.

[ANS83] ANSI. *The Programming language ADA, reference manual.* Springer-Verlag, 1983.

[ANS89] ANSI. *Programming Language – C, ANSI X3.159-1989.* ANSI, 1989.

[ASU89] A. Aho, R. Sethi et J. Ullman. *Compilateurs, principes, techniques et outils.* InterEditions, 1989.

[BBL95] P. Brun et M. Beaudouin-Lafon. A taxonomy and evaluation of formalisms for the specification of interactive systems. *Proc. Conference on Human-Computer Interaction*, p. 197–212, August 1995.

[Ber70] C. Berge. *Graphes et hypergraphes.* Dunod, 1970.

[BFL+92] L. Bass, R. Faneuf, R. Little, B. Pellegrino, S. Reed, S. Sheppard et M. Szczur. « A Metamodel for the Runtime Architecture of an Interactive System ». *ACM SIGCHI Bulletin*, 24, 1992.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire et P. Walder. Making the future safe to the past: Adding genericity to the java programmaing language. *Conf. Proc. Object-Oriented Programming: Systems, Languages, Applications (OOPSLA).* ACM SIGPLAN Notices, October 1998.

[Bro99] J. Brondeau. *Introduction à la programmation objet en Java.* Dunod, 1999.

[CKvC83] A. Colmerauer, H. Kanaoui et M. van Caneghem. « Prolog, bases théoriques et développements actuels ». *TSI*, 2:271–311, juillet-août 1983.

[Cob60] *COBOL: Initial Specifications for a Common Business Oriented Language.* Dept. Of Defense, U.S. Gov, Printing Office, 1960.

[Cou87] J. Coutaz. Pac: An object oriented model for dialog design. H.-J. Bullinger et B. Shakel, éditeurs, *Human-Computer Interaction: INTERACT'87*, p. 431–436. North-Holland, Amsterdam, 1987.

[DN66] O-J Dahl et K. Nygaard. « Simula – An Algol-based Simulation Language ». *Comm. ACM*, 9, 1966.

[Eck00] B. Eckel. *Thinking in Java.* Prentice Hall PTR, 2e édition, 2000.

[FGS90] C. Froidevaux, M. C. Gaudel et M. Soria. *Types de données et Algorithmes.* Mc Graw Hill, 1990.

[GG96] R. Griswold et M. Griswold. *The Icon Programming Language.* Peer-to-Peer Communications, 3e édition, 1996.

[GHK79] R. E. Griswold, D. R. Hanson et J. T. Korb. « The Icon programming language an overview ». *ACM SIGPLAN Notices*, 14:18–31, 1979.

[Gia91] D. Giampaolo. *Libsx v1.1 The Simple X library*, 1991.

[GJS96] J. Gosling, B. Joy et G. L. Steele. *The Java language specification.* Addison-Wesley, 1996.

[Gon95] M. Gondran. *Graphes et algorithmes.* Eyrolles, 3e édition, 1995.

[GR89] A. Goldberg et D. Robson. *Smalltalk-80, The Language.* Addison-Wesley, 2e édition, 1989.

[Hoa69] C. A. R. Hoare. « An axiomatic basis for computer programming ». *Comm. ACM*, Octobre 1969.

[Hoa72] C. A. R. Hoare. Notes on data structuring. *Structured Programming.* Academic Press, 1972.

[Hor83] E. Horowitz, éditeur. *Programming Languages, A Grand Tour.* Springer-Verlag, 1983.

[Int57] International Business Machines Corporation. *FORTRAN: an automatic coding system for the IBM 704.* IBM Corporation, New York, NY, USA, 1957.

[Knu67] D. E. Knuth. « The Remining Troublespots in Algol 60 ». *Comm. ACM*, 10, 1967.

[Knu73] D. E. Knuth. Sorting and Searching. *The Art of Computer Programming*, vol. 3. Addison Wesley, 1973.

[KP88] G. Krasner et S. Pope. « A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk -80 ». *JOOP*, 1:26–49, August/Spetember 1988.

[KR88] B. W. Kernighan et D. M. Ritchie. *The C Programming Language.* Prentice Hall, 2e édition, 1988.

[L+77] B. Liskov et al. « Abstraction Mecanisms in Clu ». *Comm. ACM*, 20:564–576, june 1977.

[M. 81] M. Shaw and others. *Alphard: Form and Content.* Springer-Verlag, 1981.

[M+89] G. Masini et al. *Les langages à objets.* InterEdition, 1989.

[MB86] M. Minoux et G. Bartnik. *Graphes algorithmes logiciels.* Dunod, 1986.

[Mey88] B. Meyer. *Conception et programmation par objets.* InterEditions, 1988.

[Mey92] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, 1992.

[Mey97] B. Meyer. *Object-oriented software construction*. Prentice Hall, 2e édition, 1997.

[NAJN75] K. V. Nori, V. Ammann, K. Jensen et Nägeli. *The Pascal (P) compiler: Implementation Notes*. Institut für Informatik, Eidgenössische Technische Hochschule, 1975.

[Nau60] P. Naur. « Report on the Algorithmic Language Algol 60 ». *Comm. ACM*, 3, 1960.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[Pfa85] G. Pfaff. User interface management systems. *Proceedings of the Seeheim Workshop*. Springer-Verlag, 1985.

[Sam69] J. E. Sammet. *Programming Languages, History and Fundamentals*. Series in Automatic Computation. Prentice-Hall, 1969.

[Sch90] D. C. Schmidt. Gperf: A perfect hash function generator. *Second USENIX C++ Conference Proceedings*, April 1990.

[Ska00] J. Skansholm. *Java From the Beginning*. Addison Wesley, 2000.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2e édition, 1986.

[W+76] W. A. Wulf et al. « An introduction to the construction and verification of Alphard programs ». *IEEE Transactions on Software engineering*, 2(3):253–264, 1976.

[Wex81] R. L. Wexelblat, éditeur. *History of Programming Languages*. Academic Press, 1981.

[Wir75] N. Wirth. *Introduction à la programmation systématique*. Masson, 1975.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs*. Printice-Hall, 1976.

[Wir85] N. Wirth. *Programming in MODULA-2*. Springer-Verlag, 2e édition, 1985.

Index

A

abstraction, 171
accès
    aux attributs, 63
    aux méthodes, 66
achèvement (prédicat d'–), 77
ACKERMANN W., 164
action
    élémentaire, 13
    structurée, 39
adressage dispersé, 275–282
affectation, 16
AL-KHOWÂRIZMÎ, 11
ALGOL 60, 8, 9
ALGOL 68, 8, 111
algorithme, 11
algorithme de
    BOYER–MOORE, 102–105, 108
    DIJKSTRA, 324–328
    EUCLIDE, 82
    FLOYD, 333
    FLOYD–WARSHALL, 321
    WARSHALL, 333
allocation mémoire, 135, 167
ALPHARD, 9, 12, 174
analyse, 6, 107
antécédent, 12
applets, 373
arbre, 221–228
    2-3-4, 260–265
    AVL, 252–260
    bicolore, 265–275
    binaire, 229–236, 286
    complet, 229
    couvrant, 319
    de jeu, 343
    équilibré, 248, 252
    ordonné, 248
    parfait, 229, 286
    quaternaire, 284
Arbre, 223–225
$Arbre_b$, 230–231
arcs, **207**, 222
arêtes, 207
ADA, 9, 176
ASCII, 27, 28, 354
attribut, 61
    partagé, 64
automate, 1
AVL, voir *arbre AVL*
AWT, 363, 364
axiomatique
    de C.A.R. HOARE, 12
    description, 172

B

backtracking, voir *rétro-parcours*
BACKUS J., 7, 8
barre de titre, 359
bicolore, voir *arbre bicolore*

boîtes à outils, 362
boolean, 27
boucle, voir *énoncé itératif*
BOYER, voir *algorithme de* BOYER–MOORE
byte, 22

C

C, XVI, 9, 20, 47, 88, 89, 93, 167
C++, 9, 176
callbacks, voir *fonctions de rappel*
carré magique, 116
cast, voir *conversions de type*
catch, 137
chaînage externe, 280
chaînes de caractères, 93
char, 28
chemin
  dans un arbre, 222
  dans un graphe, 209
  eulérien, 334
CHU SHIH-CHIEH, 99
CHURCH A., 8, 157
class, 64
classe, 61–75
  abstraite, 129
  anonyme, 366
  héritière, 123
Clé, 240
clé, **239**, 295
client-serveur, 358
clients, 61
CLU, 9
COBOL, 8
codage, 4, 238
collision, 276
COLMERAUER A., 9
commande, voir *unité de –*
commentaires, 12, 20
compilateur, 6
complément
  à deux, 22
  à un, 22
complexité, 11, **105**
composante connexe, 319
composants logiciels, 123
compression de fichier, 238
concaténation, 93
confrontation de modèle, 9, **102**
connexité, 321
conséquent, 12
constante, 16
  (complexité), 107
constructeur, 62
conversions de type
  explicites, 34
  implicites, 34
couleur, **360**, 362, 370
coupure α–β, 346
crible d'ÉRATOSTHÈNE, **94**, 155

D

DE MORGAN, 27
déclaration d'un nom, 16
décor, 359
degré, 209
Dèque, 202
dèque, 179, **201–203**
descendant, 124
description
  axiomatique, 172
  fonctionnelle, 172
dictionnaire, voir *table*
DIJKSTRA, voir *Algorithme de –*
do, 79
documentation, 20, 48
donnée, 2, 13
double, 25
DÜRER A., 116

E

EBCDIC, 27
écriture, **15**, 143
effet de bord, 54
EIFFEL, 9, 12, 66, 136, 174, 176
en-tête, 48
énoncé
  choix, 40
  composé, 39
  conditionnel, 40
  itératif, 77
  répéter, 79
  si, 42
  tantque, 78
entrée standard, 13
énumération, 192
environnement
  logiciel, 1
  matériel, 1

équipements externes, 2
ÉRATOSTHÈNE, 94, 95, 155
EUCLIDE, 82, 85
EULER L., 334
évaluation, 32
événements, 358, **363**
exception, 135
exécution
    parallèle, 39
    séquentielle, 39
expressions, 31
extends, 132

F

fenêtre, voir *système de fenêtrage*
fermeture transitive, 321
feuilles, 222
FIBONACCI, 159
fichier, 141
    de texte, 151
    séquentiel, **141**, 313
FIFO, voir *file*
file, 179, **199–201**
File, 199–200
File–Priorité, 285–286
fils, 222
final, 19
finitude, 80
float, 25
flot, 145
FLOYD, voir *algorithme de –*
flux, voir *flot*
focus, 359
fonction, 14, **47**
fonctionnelle (description), 172
fonctions
    d'adressage, 275
    de rappel, 363
    secondaires, 278
for, 98
Forêt, 223–225
FORTRAN, 7, 8, 111
fournisseur, 61
frères, 222
FSF, 4
fusion, 147, 313

G

garbage-collector, 208
GAUSS C. F., 119, 337
générateur aléatoire, 147
généricité, 133, 176
géométrie, 360
gestionnaire de fenêtres, 3, **358**
grands entiers, 107
graphe, 207–219
    connexe, 209
    d'héritage, 130
    non orienté, 207
    orienté, 207
    sans cycle, 331
    simple, 208
    valué, **208**, 210, 212, 215
Graphe, 209–210
GRISWOLD R., 9

H

hash-coding, voir *adressage dispersé*
heapsort, voir *tri en tas*
héritage, 123, 124
    multiple, 130
    simple, 130
HÉRON L'ANCIEN, 85
HILBERT (courbes de –), 165
HOARE C.A.R, 12, 171, 308, 311
HORNER W. (schéma de –), 99, 100, 108
HTML, 20, 373, 375
HUFFMAN (code de –), 238
huit reines, 337

I

IBM, 7, 27
ICON, 9, 32
identificateur
    de constante, 16
    de fonction, 48
    de procédure, 48
    de variables, 17
IEEE, 24, 25
if, 43
implantation, 174
    d'*Arbre*, 225
    d'*Arbre*$_b$, 231
    de *Dèque*, 202
    de *File*, 200
    de *Forêt*, 225
    de *Graphe*, 211
    de *Liste*, 181
    de *Pile*, 196

de *Table*, 242
énumération, 193
import, 134
indexation, 87
indice, 87
instances, 62
int, 22
interclassement, 313
interface, 133
graphique, 357–375
textuelle, 353
utilisateur, 353, 355
interpolation, 282
interprète, 6
de commandes, 3
invariant
de boucle, 77
de classe, **68**, 136
ISO, 27, 28
ISO-8859, 28

J

JAVA, 9
javadoc, 20
jeux de stratégie, 342

K

KNUTH D., 8, 306
KRUSKAL, 321

L

labyrinthe, 335, 350
langage
à objets, 9
d'assemblage, 4
de programmation, 4
intermédiaire, 6
machine, 4
non typé, **21**, 127
typé, **21**, 128
lecture, **13**, 144
lexical, 5
liaison dynamique, 129
LIFO, voir *pile*
linéaire (complexité), 107
lisibilité, 48
LISP, 7–9, 31
LISP 2, 8
liste, 179–192
d'adjacence, 211, **214**, 227
non ordonnée, 242
ordonnée, 244
Liste, 180–181
localisation, 48
logarithmique (complexité), 107
logiciel, 3
libre, 4
long, 22
ŁUKASIEWICW JAN, 31

M

main, 19, 68
matrice, 111
creuse, 205
d'adjacence, 211
produit, 115
symétrique, 114
MCCARTHY J., 7, 8
médiane, 310, 317
mémoire
caches, 2
centrale, 1
principale, 1
récupérateur de –, 208
secondaires, 2
menus, 359
méthode, 61, 65
abstraite, 129
de GAUSS, 119
de SIMPSON, 109
des rectangles, 108
redéfinition, 126
surcharge, 66
MEYER B., 61, 66, 68, 131
MinMax, 342
modèle
ARCH, 357
de SEEHEIM, 356
MVC, **356**, 364
MODULA, 9
monotonies, 313
MOORE, voir *algorithme de* BOYER–MOORE
MORGENSTERN O., 342
multigraphe, 208

N

new, 64

Nœud, 223
nœuds, 222
notation
  $\mathcal{O}$, 106
  polonaise, 31
  postfixée, 31
  préfixée, 31
noyau fonctionnel, 355
`null`, 90

O

objets, 13
objets dynamiques, 61
occurrence, 102
octet, **2**, 145
opérandes, 31
opérateurs, 31
opérations abstraites, 171
ordinateur, 1

P

père, 222
paramétrage, 48
paramètre, 48
  effectif, 14, **50**
  formel, 48
parcours
  d'arbre, 228
  d'arbre binaire, 234
  de graphe, 215–219
  de liste, 193
partition, voir *tri rapide*
PASCAL, 9, 16, 34, 47, 88, 93, 111, 151, 167
pattern matching, voir *confrontation de modèle*
PEANO G., 173
périphériques, 2
permutation, 169, 295
pgcd, 82
pile, 179, **195–199**
Pile, 196
pivot, 308
pixels, 360
PL/I, 8
plus court chemin, voir *algorithme de* DIJKSTRA
pointeurs, 167
polices de caractères, 28
polymorphisme, 128
polynôme, 99, 204
portabilité, 7
post-condition, 12
pré-condition, 12
preuve, 171
PRIM, 321
priorité, 32
`private`, 64
procédure, 14, **47**
produit de matrices, 115
programmation
  contractuelle, 69
  descendante, 10
  objet, 9
programme, 4
PROLOG, 9
promotions, 34
prototype, 48
`public`, 64
puits (d'un graphe), 220

Q

quadratique (complexité), 107
quicksort, voir *tri rapide*

R

récupérateur de mémoire, 208
recherche, 239
  auto-adaptative, 244
  de chaînes, 102
  dichotomique, 246–248
  en arbre, 249
  linéaire, **242**, 245
  par interpolation, 282
récursivité, 157
  des actions, 158
  des objets, 166
redéfinition, voir *méthodes*
référence, 62
registres, 2
règles
  de déduction, 12
  de priorité, 32
résultat, 13
rétro-parcours, 335–342
`return`, 68
réutilisabilité, 10, **123**
RITCHIE D., 9
rotations, 253–256

S

SEDGEWICK R., 316
sémantique, 5
sentinelle, 243
séquence, 179
SHELL D.L., 305
short, 22
signature, 48
SIMPSON, 109
SIMULA, 9
SL5, 9
SMALLTALK, 9, 356
SNOBOL, 9, 93
sommets, 207
sortie standard, 15
sous-arbres, 222
sous-classes, 132
sous-programme, 47
stable, voir *tri stable*
STALLMAN R., 4
static, 64, 68
STRAHLER (nombre de –), 237
STRASSEN, 116
stream, voir *flot*
structuration, 48
structure de données, 171
structures linéaires, 179
succ, 193
successeur, 179
super, 132
super-classe, 132
surcharge, voir *méthode*
Swing, 364
switch, 41
syntaxique, 5
système
  d'exploitation, 3
  de fenêtrage, 354
  interactif, 357

T

table, 239
  d'adressage dispersé, 275
  de vérité, 26
Table, 240
tableau, 87–92
  à plusieurs dimensions, 111–118
  circulaire, 184
tas, 286–293, 298, 328
temps partagé, 3
this, 64
this(), 68
throw, 138
throws, 138
`toString`, 71
traducteur, voir *compilateur*
transmission
  par résultat, 51
  par valeur, 51
tri, 295–316
  à bulles, 307
  en tas, 298, 302
  externe, 295, **313–316**
  fusion, 316
  fusion naturelle, 313
  idiot, 316
  insertion séquentielle, 302
  interne, 295
  rapide, 308–311
  sélection directe, 298
  SHELL, 305–307
  stable, 316
  topologique, 328–332
  topologique inverse, 331
try, 137
type, 21
  abstrait algébrique, 171
  booléen, 26
  caractère, 27
  élémentaire, 21
  entier, 22
  énuméré, 29
  fichier, 141
  intervalle, 29
  réel, 23
  simple, 21
  structuré, 61

U

UNICODE, 28, 29, 46, 93, 94, 104, 152, 155
unité
  arithmétique et logique, 2
  centrale de traitement, 2
  d'échanges, 2
  de commande, 2

V

variable, 15
VON NEUMANN J., 1, 342

W

WARSHALL, voir *algorithme de –*
WEB, 4, 9, 373
while, 79
widget, 362
WIRTH N., 9, 11

X

XEROX, 353

048463 - (I) - (2) - OSB 80° - AUT - JME

Achevé d'imprimer sur les presses de
SNEL Grafics sa
rue Saint-Vincent 12 – B-4020 Liège
Tél +32(0)4 344 65 60 - Fax +32(0)4 341 48 41
juin 2004 – 31989

Dépôt légal : juillet 2004
Dépôt légal de la 1re édition : septembre 2000

Imprimé en Belgique